

Java 2 图形设计

卷II: SWING

Graphic Java 2 Mastering the JFC Volume II:SWING,
3rd Edition

(美) David M. Geary 著 李建森 蒋欣军 龚尧莞 等译



机械工业出版社
China Machine Press

PTR
PH

Java 2 图形设计

卷II: SWING

- 专业图形用户界面工具
- 用SWING开发的设计思想
- 最全面的SWING工具参考书
- 本书作者是开发Java Soft的Java Management API 用户界面工具包的首席工程师
- 附带光盘包含全部实例程序代码

- 《Windows 2000 编程技术内幕》
- 《Visual C++ 6.0 编程实例与技巧》
- 《中文FrontPage 2000 开发人员指南》
- 《Visual Basic 6 开发人员指南》
- 《Visual Basic 6 数据库开发人员指南》
- 《Visual Basic 6 数据库访问技术》
- 《Perl 5 编程详解》
- 《Delphi 4 编程技术内幕》
- 《SQL Server 7 编程技术内幕》
- 《Visual Basic 6 开发人员参考手册》
- 《Visual C++ MFC 编程实例》
- 《Visual C++ MFC 扩展编程实例》
- 《MFC Visual C++ 6 编程技术内幕》
- 《Java 2 核心技术 卷I: 基础知识》
- 《Java 2 图形设计 卷I: AWT》
- 《Java 2 图形设计 卷II: SWING》
- 《SQL-3 参考大全》
- 《Windows WDM 设备驱动程序开发指南》

- 万能语言
- 一流品质

无论您是初学者还是高级用户, 机械工业出版社都有适合您口味的图书



《Java 编程思想》

书号: ISBN7-111-07064-X TP·1011
页数: 666 页
定价: 60.00 元



《Java1.2 24学时学习教程》

书号: ISBN 7-111-06995-1 TP·980
页数: 262 页
定价: 28.00 元



《软件工程: Java语言实现》

书号: ISBN 7-111-07355-X TP·1147
页数: 396 页
定价: 38.00 元

ISBN 7-111-07774-1

封面设计: 胡京湘



附赠



满园春相似
惟我花不同



北京市西城区百万庄大街22号 100037

购书热线: (010) 8006100280

www.hzbook.com

ISBN 7-111-07774-1/TP·1336

定价: 108.00 元

软件开发技术丛书

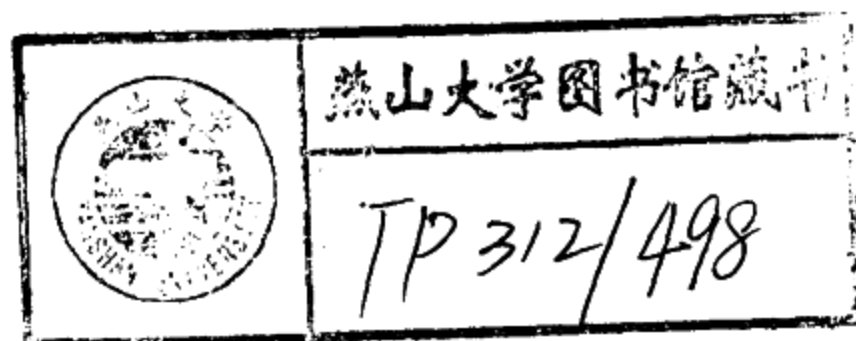
Java 2 图形设计

卷 II : SWING

(美) David M. Geary 著

李建森 蒋欣军 龚尧莞 等译

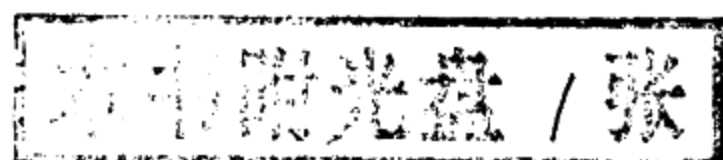
万 华 审校



0268166



机械工业出版社
China Machine Press



Swing 是一流的 Java 图形用户界面开发工具。本书详细介绍了 Swing 的设计思想、体系结构、使用技巧,内容丰富、深入细致、分析透彻。本书用大量实例代码介绍了每个组件的用法,使初学者能很快入门;用大量图示分析了 Swing 组件的特点、结构及相互关系,使有经验的编程人员能高效利用 Swing 的强大功能。本书对掌握 Swing 技术提供了最全面的参考。

David M. Geary: Graphic Java 2 Mastering the JFC Volume II : Swing, 3rd Edition.

Authorized translation from the English language edition published by Prentice Hall PTR.

Copyright © 1999 by Sun Microsystems, Inc..

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2000 by China Machine Press.

本书中文简体字版由美国 Prentice Hall PTR 公司授权机械工业出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书的任何部分。

版权所有,侵权必究。

本书版权登记号:图字:01-1999-2353

图书在版编目 (CIP) 数据

Java 2 图形设计 卷 II: SWING/ (美) 吉瑞 (Geary, D.M.) 著; 李建森等译.
—北京: 机械工业出版社, 2000.2

(软件开发技术丛书)

书名原文: Graphic Java 2 Mastering the JFC Volume II: SWING, 3rd Edition

ISBN 7-111-07774-1

I. J… II. ①吉…②李… III. JAVA 语言-程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (1999) 第 56459 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 吴 怡 瞿静华

北京京丰印刷厂印刷·新华书店北京发行所发行

2000 年 2 月第 1 版第 1 次印刷

787mm × 1092mm 1/16 · 62.25 印张

印数: 0 001 ~ 5 000 册

定价: 108.00 元 (附光盘)

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换

译者序

随着计算机技术和应用的普及，图形用户界面的开发愈加受到计算机应用开发人员的重视。Swing 是 Java 图形用户界面开发工具包的重要组成部分，它在原来的 Java 用户界面开发工具包 AWT（抽象窗口工具包）的基础上进行了重大改进，为 Java 应用程序开发人员提供了功能强大、业界一流的图形用户界面开发工具，使 Java 图形用户界面的开发更为实用、简单和方便。Swing 突出地体现了 Java 的简单、跨平台、面向对象和可移植等特性，必将进一步推动 Java 应用的普及。

本书是关于 Swing 的权威性参考书。作者 David M. Geary 是开发 Java Soft 的 Java Management API 用户界面工具包的首席工程师。本书为 Swing 的初学者提供了大量的源代码实例，以便初学者迅速掌握 Swing 的体系结构，还为有经验的编程人员介绍了如何充分地利用 Swing 的强大功能。本书介绍了使用 Swing 的实践经验，提供了创建带有复杂图形用户界面的 Java 应用程序所必需的资源。因此，本书既适用于 Swing 的初学者，也适用于有经验的 Swing 开发人员。

本书分三大部分。第一部分介绍了 Swing 小应用程序与应用程序、Swing 与多线程、Swing 的“模型视图控制器”结构和插入式界面样式等 Swing 基础知识；第二部分详细介绍了 Swing 的所有关键组件，包括按钮、标签、进度条、滑杆、窗口与对话框、内部窗体和桌面窗格、菜单和工具条、颜色和文件选取器等，并列举了说明它们的使用方法的大量实例；第三部分附录，介绍了插入式界面样式的常量，为读者提供方便。

参加本书翻译工作的人员有：李建森、蒋欣军、龚尧尧、王彬星、卢庆龄、扬朝红、崔伟宁、李颖、周兵、张宏军、杨明立、陈华、陈豫、谢冰梅、张志弘、李玉为、李建新、杨菊梅、陈英武、邱立平、何春强、师彦彬、汪芸、赵大力、王臣太、牟志全、牛立全、王庆辰、李清、王伟、钟家杰、万志龙、吴寿虎、于云、师帅帅、宋冰清、刘一梅、钱前、黄诤等。全书由李建森统稿，万华审校。

由于本书篇幅庞大，译者水平有限，加之时间仓促，书中错漏之处在所难免，敬请广大读者批评指正。

1999 年 12 月

015/01/01

序

不要问我计算机用户界面设计出现多少年了。有多少人拆开并重装过电传打字机？有多少人还记得什么时候电传打字机上有一个“T”字母（由于它是一个公司及其产品的名称）？令人庆幸的是，这些都已成过眼烟云。过去计算机少，任务多，所以，用户界面不得不很简单，用户必须经过训练。现在的情况完全不同了。在许多现代应用程序中，应用程序的设计要求受到了用户界面的设计要求的挑战；或者说，用户界面的设计要求甚至超过了应用程序本身的设计要求。随着基于语音和人工智能的计算机用户界面的出现，这种情况将会更加明显。

在当今世界中，如果考察一下开发人员必须使用的各种不同的编程界面，那么就不难发现，与用户交互的界面是最重要的，且是使用最广泛的。现在，用户界面不再是强加在应用程序上的附属品，而且常常是应用程序开发的核心。正是在这种背景下，我们开发了 Swing。Swing 是复杂的用户界面开发工具，是一个功能全面的、在业界具有竞争力的开发系统。

我参与了 Swing 的早期开发工作。在建立 Swing 构件（大多数早已不存在了）的过程中，在与开发团队一起工作的过程中，我得到了很多乐趣。从那时起，作为 Swing 的一个用户，我也十分开心。希望读者们也能与我一样从中获得无尽的乐趣。

不要被本书的厚度吓倒。Swing 用户界面工具包是很复杂的，其中有很多技巧。在 Swing 的设计中，一方面要满足业界对大量功能的需求，另一方面要满足 Java 编程的简单性，因为这是 Java 最令人钦佩的特性之一；这是很难取舍的。作为开发人员，处理这个矛盾的关键是理解简单与容易之间的区别。Swing 工具包不是简单的，对 Swing 的功能需求很多。但 Swing 又很容易使用，因为其中有许多简捷的方法（不妨看一下 JOptionPane 类），并且普通开发人员可以忽略 Swing 的大多数特性。

请首先概览一下本书的内容，以便了解需要掌握 Swing 的哪些内容。然后再深入进去，以概览为向导，深入到需要细致研究的地方。

Sun Microsystem 公司副总裁
James Gosling

前 言

本书是我一年多来全部热情的结晶。我力争使本书成为最全面、最准确及最深入的 Swing 技术书籍。当然，我是否达到了目的，最终将由读者们作出自己的判断。

在开始写作之前，我花了很多时间来设计本书，希望它既适用于具有一定 Swing 经验的开发人员，又适用于 Swing 的初学者。作为一个使用过大量 GUI 对象的开发人员，我认识到，学好一个 GUI 结构的最快方法是学习那些能够说明特定概念的代码实例。因此，代码实例是本书的基础，本书详细讨论了大约 300 个代码实例。

但是，仅参考代码实例对那些已经掌握了大量 Swing 知识的开发人员来说是不够的。因此，在讨论每一个 Swing 组件后都带有一个组件总结。组件总结中包含了类图、对组件属性和事件的详细介绍以及类总结（类总结讨论由该组件实现的 public 和 protected 方法），以便开发人员对组件有深入的理解。

本书的第一部分探讨了 Swing 的基础；第二部分详细介绍了 Swing 的组件，从标签和按钮到表、树和文本包，用大量的代码实例对每一个 Swing 组件都进行了介绍。例如，对表组件的介绍用了 25 个代码实例，为了说明如何充分利用树组件使用了 20 个代码实例；第三部分附录介绍了插入式界面样式的常量。

本书的读者

本书假定读者很好地掌握了 Java 语言，包括该语言中新近增加的内部类等成分。本书还假定读者初步了解 AWT；特别是代理事件模型和 Swing 所有组件的建立基础 Component 类和 Container 类。要对 AWT 体系结构和组件的彻底研究，请参见《Java 2 图形设计，卷 I：AWT》。

如何使用本书

在深入学习 Swing 组件之前，理解 Swing 的模型视图控制器（MVC）结构和插入式界面样式等基本概念是有好处的。本书第 3 章讨论了 MVC，第 7 章介绍了插入式界面样式。理解 JComponent 类提供的服务也是很重要的。JComponent 是所有轻量（lightweight）Swing 组件的最终超类。JComponent 在第 4 章讨论。

组件总结

对每个 Swing 组件的介绍都引用了说明各种组件特性的大量代码实例。在代码实例之后还提供了组件总结。组件总结是以一个组件模型表开始的，然后是 UI 代表、绘制器和编辑器，以及由该组件激发的事件。如果该组件是用来替代一个 AWT 组件的，则还列出了替换的 AWT 组件。

类图

表示了一个组件与其他对象保持的静态关系。

属性

一个组件的属性表包括属性名、属性的数据类型，以及一个属性是否是布尔型的、关联的（当改变这个属性时，激发一个属性改变事件）、受约束的（可以否决对属性的修改）、简单的（当改变这个属性时，不激发任何事件）或索引的（这个属性可以用一个参数，通常是一个整数来访问）。

有些属性是可以在初始化一个组件时指定。组件可以为属性提供“设置”和“获取”方法。指定一个属性的方式列在属性表的访问栏中。属性表还包含一个属性缺省值栏。属性表后面还有对表中列出的每一个属性的简短描述。

事件

本书为读者提供了一些代码实例，它们说明了组件的事件处理方法。例如，第 20 章“树”中提供了五个代码实例，它们说明了树鼠标、树编辑、树选取和树扩展事件的处理方式。

类总结

每一个组件总结都以一个类总结作为结束。类总结对该组件的构造方法和其他方法进行了描述。

Swing 中的程序错误

在质量方面，Swing 已经走过了漫漫长途。早期发布的 beta 版本含有大量的程序错误，其中的许多错误已相继被纠正了。但是，与许多软件一样，Swing 仍然含有程序错误。在整本书中，我都试图尽可能多地指出 Swing 中的错误，以便使开发人员避免调试代码带来的挫折感，因为调试的结果仅仅是发现由于 Swing 的错误引起的问题是没有必要的。

记住本书介绍的是 Swing 1.1FCS，这也是很重要的。当本书即将出版时，Swing 的 1.1.1 版本也已发布。1.1.1 版几乎就是一个纠错版本。因此，可以肯定，当本书放上书架时，本书中指出的错误已经被纠正了。

本书的 CD-ROM

本书所带的 CD-ROM 盘中包含了如下内容：

- JDK1.1.7 和 Swing1.1.1
- JDK1.2 和 Swing1.1FCS
- 本书中的代码实例

Swing 与 JDK

如前所述，Swing1.1.1 几乎是一个纠错版。Swing1.1.1 不能与 JDK1.2 一起使用，因此，本书附带 CD-ROM 中包含了 JDK1.1.7。Swing1.1FCS 可以与 JDK1.1.7 和 JDK1.2 一起使用。

本书中的代码实例

本书中的实例几乎都可以在随带的 CD-ROM 中找到。CD-ROM 中，本书的每一章都有一个目录，每一个例子对应相应章的子目录。这种组织方式使读者容易在 CD-ROM 中找到例子。每个例子都已编译好，可以直接运行。如果其中的例子展示了程序错误或与书中的相应程序清单

比较有所修改的话，该代码实例的目录中会包含 README.txt 文件。

这个 CD-ROM 盘中还带有《Java2 图形设计，卷 I：AWT》（该书已由机械工业出版社出版——编者注）所带的 GridBayLab 应用程序的两个版本。其中一个版本使用外部窗口，另一个例子则使用 Swing 内部窗体。这个应用程序是一个完整的 Swing 应用程序的例子，说明了 Swing 内部窗体的使用方法。另外，用这个应用程序可以探讨 GridBayLayout 布局管理器的复杂性。

小应用程序与应用程序

本书中讨论的大多数代码实例是小应用程序，但也有相当一部分例子是应用程序。如果某程序以应用程序形式而不以小应用程序形式实现，是由于该程序处理了文件，或者显示了对话框。小应用程序访问文件的能力有限。另外，在 JDK1.2 下，小应用程序显示的对话框中包含了一个警示字符串。有时，某程序段以小应用程序实现仅仅是为了例子的多样性。本书中讨论的几乎所有不处理文件的应用程序都能够很容易地以小应用程序的形式改写。

本书英文原书书号：ISBN 0-13-079667-0

英文原书出版社网站地址：www.phptr.com

目 录

译者序	
序	
前言	
第一部分 Swing 基础	1
第 1 章 简介	1
1.1 Swing 的历史	1
1.2 轻量组件与重量组件的比较	2
1.3 Swing 组件	2
1.3.1 AWT 的替代组件	3
1.3.2 Swing 增加的组件	3
1.4 J 组件	4
1.5 Swing 包概览	6
1.6 Swing 与 AWT	8
1.7 开始学习	9
1.8 Swing 资源	11
1.9 本章回顾	11
第 2 章 Swing 的基本知识	13
2.1 小应用程序与应用程序	13
2.1.1 小应用程序	13
2.1.2 JApplet 类	14
2.1.3 应用程序	17
2.1.4 JFrame 类	18
2.1.5 小应用程序/应用程序的组合	19
2.2 GJApp	21
2.3 混合使用 Swing 组件和 AWT 组件	23
2.3.1 层序	23
2.3.2 Swing 弹出式菜单	26
2.3.3 滚动	28
2.3.4 内部窗体	30
2.4 Swing 和线程	31
2.4.1 Swing 单线程设计的结果	32
2.4.2 SwingUtilities 类的 invokeLater 和 invokeAndWait 方法	32
2.5 本章回顾	40
第 3 章 Swing 组件的体系结构	41
3.1 典型的“模型-视图-控制器”体系 结构	41
3.1.1 插入式视图和控制器	41
3.1.2 视图更新	42
3.2 Swing MVC	42
3.2.1 Swing 组件	44
3.2.2 静态认识	45
3.2.3 动态认识	46
3.2.4 模型	48
3.2.5 UI 代表	58
3.2.6 组件 UI 的案例	59
3.2.7 监听器	68
3.3 本章回顾	72
第 4 章 JComponent 类	73
4.1 JComponent 类概览	73
4.1.1 边框	73
4.1.2 可访问性	74
4.1.3 双缓存	75
4.1.4 调试图形	75
4.1.5 自动滚动	76
4.1.6 工具提示	77
4.1.7 键击处理和客户属性	77
4.2 JComponent 类结构	77
4.2.1 Swing 组件是 AWT 容器	78
4.2.2 最小尺寸、最大尺寸和首选 尺寸	78
4.3 绘制 JComponent 组件	81
4.3.1 Swing 组件中的定制绘制	82
4.3.2 在 AWT 组件中重载绘制方法	82
4.3.3 在 Swing 组件中重载绘制方法	83
4.3.4 paint、repaint 和 update 方法	85
4.3.5 validate、invalidate 和 revalidate 方法	85
4.3.6 不透明组件与透明组件的比较	86
4.3.7 立即绘制 Swing 组件	88
4.4 双缓存	89
4.5 调试图形	96
4.6 自动滚动	100
4.7 工具提示	104
4.7.1 基于鼠标位置的工具提示	105
4.7.2 工具提示的首选位置	107

4.7.3 定制工具提示的行为	108	6.7.5 状态编辑	193
4.7.4 定制工具提示的界面样式	109	6.8 本章回顾	197
4.8 键击处理	109	第7章 插入式界面样式	198
4.9 客户属性	113	7.1 界面样式结构	198
4.10 焦点管理	116	7.1.1 界面样式	199
4.10.1 JComponent 的焦点属性	116	7.1.2 界面样式缺省值	204
4.10.2 焦点管理器	119	7.1.3 UI 管理器	208
4.11 支持可访问性	120	7.1.4 UI 资源	213
4.12 本章回顾	122	7.2 Java 界面样式	217
第5章 边框、图标和动作	123	7.2.1 客户属性	217
5.1 边框	123	7.2.2 主题	222
5.1.1 边框和边衬	123	7.3 附加 UI	223
5.1.2 Swing 的边框类型	124	7.4 本章回顾	226
5.1.3 不透明与透明之间的比较	128	第二部分 Swing 组件	227
5.1.4 边框包	129	第8章 标签与按钮	227
5.1.5 边框接口	130	8.1 JLabel 与 JButton	227
5.1.6 AbstractBorder 类	130	8.2 JLabel	229
5.1.7 边框库——共享边框	131	8.2.1 内容排列	230
5.1.8 替换内置边框	132	8.2.2 文本的位置	233
5.1.9 实现定制边框	133	8.2.3 图标/文本间隙	235
5.2 图标	135	8.2.4 许可状态	236
5.2.1 把图标与组件相关联	136	8.2.5 JLabel 属性	238
5.2.2 在组件中共享图标	137	8.2.6 JLabel 事件	239
5.2.3 图像图标	140	8.2.7 JLabel 类总结	239
5.2.4 动画的图像图标	142	8.3 按钮	241
5.3 动作	143	8.4 JButton	243
5.3.1 作为控制中心点的动作	145	8.4.1 JButton 属性	245
5.3.2 动作常量	149	8.4.2 JButton 事件	248
5.4 本章回顾	150	8.4.3 JButton 类总结	251
第6章 实用工具	152	8.4.4 AWT 兼容	260
6.1 计时器	152	8.5 本章回顾	261
6.2 事件监听器列表	158	第9章 反转按钮、复选框和单选钮	262
6.3 Swing 实用工具	160	9.1 JToggleButton 类	262
6.4 Swing 常量	166	9.1.1 JToggleButton 属性	263
6.5 BoxLayout 和 Box 类	167	9.1.2 JToggleButton 事件	263
6.5.1 BoxLayout 类	167	9.1.3 JToggleButton 类总结	264
6.5.2 Box 类	169	9.1.4 AWT 兼容	266
6.6 进度监视器	172	9.2 按钮组	266
6.6.1 ProgressMonitor	172	9.3 复选框	267
6.6.2 ProgressMonitorInputStream	176	9.3.1 JCheckBox 属性	270
6.7 撤消/重复	179	9.3.2 JCheckBox 事件	270
6.7.1 一个简单的撤消/重复样例	181	9.3.3 JCheckBox 类总结	270
6.7.2 UndoableEditSupport	185	9.4 单选钮	272
6.7.3 组合编辑	187	9.4.1 JRadioButton 属性	275
6.7.4 UndoManager	192	9.4.2 JRadioButton 事件	275

9.4.3 JRadioButton 类总结	275	10.9.5 AWT 兼容	345
9.4.4 AWT 兼容	276	10.10 JToolBar	346
9.5 本章回顾	276	10.10.1 滚过式工具条	349
第 10 章 菜单和工具条	278	10.10.2 在工具条中使用动作	350
10.1 菜单、菜单栏和工具条	278	10.10.3 浮动工具条	352
10.2 菜单和弹出式菜单	282	10.10.4 位置固定的工具提示	353
10.3 JMenuItem	283	10.10.5 JToolBar 属性	355
10.3.1 菜单项快捷键和助记符键	286	10.10.6 JToolBar 事件	356
10.3.2 JMenuItem 属性	289	10.10.7 JToolBar 类总结	356
10.3.3 JMenuItem 事件	289	10.10.8 AWT 兼容	357
10.3.4 JMenuItem 类总结	292	10.11 本章回顾	357
10.3.5 AWT 兼容	294	第 11 章 进度条、滑杆和分隔条	358
10.4 JCheckBoxMenuItem	295	11.1 JProgressBar	358
10.4.1 JCheckBoxMenuItem 属性	296	11.1.1 进度条与线程	359
10.4.2 JCheckBoxMenuItem 事件	296	11.1.2 JProgressBar 属性	363
10.4.3 JCheckBoxMenuItem 类总结	297	11.1.3 JProgressBar 事件	364
10.4.4 AWT 兼容	298	11.1.4 JProgressBar 类总结	366
10.5 JRadioButtonMenuItem	298	11.1.5 AWT 兼容	368
10.5.1 JRadioButtonMenuItem 属性	299	11.2 JSlider	368
10.5.2 JRadioButtonMenuItem 事件	299	11.2.1 填充的滑杆	368
10.5.3 JRadioButtonMenuItem 类 总结	303	11.2.2 滑杆间隔标记	369
10.5.4 AWT 兼容	304	11.2.3 滑杆标签	373
10.6 JMenu	304	11.2.4 反转滑杆值	375
10.6.1 动态修改菜单	305	11.2.5 滑杆的外延值	376
10.6.2 右拉式菜单	309	11.2.6 JSlider 属性	378
10.6.3 JMenu 属性	311	11.2.7 JSlider 事件	379
10.6.4 JMenu 事件	313	11.2.8 JSlider 类总结	380
10.6.5 JMenu 类总结	314	11.2.9 AWT 兼容	382
10.6.6 AWT 兼容	319	11.3 JSeparator	382
10.7 菜单元素	320	11.3.1 分隔条与框	385
10.8 JPopupMenu	324	11.3.2 JSeparator 属性	387
10.8.1 弹出式菜单触发器	326	11.3.3 JSeparator 事件	387
10.8.2 轻量/中量/重量弹出式 菜单	327	11.3.4 AWT 兼容	388
10.8.3 弹出式菜单调用者	328	11.4 本章回顾	388
10.8.4 JPopupMenu 属性	331	第 12 章 轻量容器	389
10.8.5 JPopupMenu 事件	333	12.1 JPanel	389
10.8.6 JPopupMenu 类总结	334	12.1.1 JPanel 的属性	391
10.8.7 AWT 兼容	338	12.1.2 JPanel 的事件	391
10.9 JMenuBar	338	12.1.3 JPanel 类总结	391
10.9.1 菜单栏菜单和组件	339	12.1.4 AWT 兼容	392
10.9.2 JMenuBar 属性	343	12.2 JRootPane	392
10.9.3 JMenuBar 事件	344	12.2.1 RootPaneContainer 接口	392
10.9.4 JMenuBar 类总结	344	12.2.2 玻璃窗格	393
		12.2.3 内容窗格	397
		12.2.4 JRootPane 属性	401

12.2.5 JRootPane 事件	402	13.4.5 JScrollBar 类总结	487
12.2.6 JRootPane 类总结	405	13.4.6 AWT 兼容	488
12.2.7 AWT 兼容	406	13.5 本章回顾	488
12.3 JLayeredPane	406	第 14 章 窗口和对话框	490
12.3.1 回顾轻量组件的层序	407	14.1 JWindow	490
12.3.2 为组件分配层	409	14.1.1 JWindow 属性	494
12.3.3 指定同一层中组件的位置	411	14.1.2 JWindow 类总结	495
12.3.4 使用拖动层	415	14.1.3 AWT 兼容	496
12.3.5 JLayeredPane 属性	420	14.2 JDialog	496
12.3.6 JLayeredPane 类总结	420	14.2.1 JDialog 属性	501
12.3.7 AWT 兼容	422	14.2.2 JDialog 类总结	501
12.4 JTabbedPane	422	14.2.3 AWT 兼容	503
12.4.1 选项卡的位置	424	14.3 JOptionPane	503
12.4.2 JTabbedPane 的属性	429	14.3.1 内部窗体	506
12.4.3 JTabbedPane 事件	430	14.3.2 用 JOptionPane 静态方法创建 对话框	507
12.4.4 JTabbedPane 类总结	431	14.3.3 消息对话框	509
12.5 JSplitPane 类	433	14.3.4 确认对话框	514
12.5.1 JSplitPane 属性	439	14.3.5 输入对话框	517
12.5.2 JSplitPane 事件	439	14.3.6 选项对话框	521
12.5.3 JSplitPane 类总结	440	14.3.7 JOptionPane 属性	525
12.5.4 AWT 兼容	442	14.3.8 JOptionPane 事件	527
12.6 本章回顾	442	14.3.9 JOptionPane 类总结	530
第 13 章 滚动	443	14.3.10 AWT 兼容	538
13.1 JViewport	443	14.4 本章回顾	538
13.1.1 拖动视口中的视图	446	第 15 章 内部窗体和桌面窗格	539
13.1.2 使用 scrollRectToVisible 方法	448	15.1 JInternalFrame	539
13.1.3 JViewport 属性	452	15.1.1 JInternalFrame 属性	541
13.1.4 JViewport 事件	452	15.1.2 JInternalFrame 事件	543
13.1.5 JViewport 类总结	455	15.1.3 AWT 兼容	552
13.1.6 AWT 兼容	457	15.2 JDesktopPane	553
13.2 JScrollPane	457	15.2.1 JDesktopPane 属性	558
13.2.1 滚动窗格的头部	458	15.2.2 JDesktopPane 事件	558
13.2.2 滚动窗格的角部	464	15.2.3 JDesktopPane 类总结	558
13.2.3 JScrollPane 属性	467	15.2.4 AWT 兼容	559
13.2.4 JScrollPane 事件	468	15.3 DesktopManager	559
13.2.5 JScrollPane 类总结	468	15.4 本章回顾	567
13.2.6 AWT 兼容	472	第 16 章 选取器	568
13.3 Scrollable 接口	473	16.1 JFileChooser	568
13.4 JScrollBar	476	16.1.1 文件选取器类型	572
13.4.1 使用 Swing 的 JScrollBar 类进 行手动滚动	476	16.1.2 可访问组件	576
13.4.2 块增量和单元增量	479	16.1.3 过滤文件类型	580
13.4.3 JScrollBar 属性	484	16.1.4 文件视图	589
13.4.4 JScrollBar 事件	485	16.1.5 多文件选取	593
		16.1.6 JFileChooser 属性	596

16.1.7 JFileChooser 事件	599	19.2.2 TableModel 接口	710
16.1.8 JFileChooser 类总结	603	19.2.3 AbstractTableModel	710
16.1.9 AWT 兼容	608	19.2.4 DefaultTableModel	713
16.2 JColorChooser	608	19.2.5 表格模型、缺省绘制器 和缺省编辑器	718
16.2.1 在对话框中显示颜色 选取器	610	19.3 表格列	721
16.2.2 定制颜色选取器	614	19.3.1 列调整大小模式	722
16.2.3 JColorChooser 属性	624	19.3.2 列宽度	725
16.2.4 JColorChooser 事件	624	19.4 表格列模型	731
16.2.5 JColorChooser 类总结	624	19.4.1 DefaultTableColumnModel 类	732
16.2.6 AWT 兼容	626	19.4.2 列边距	733
16.3 本章回顾	626	19.4.3 隐藏列	736
第 17 章 列表	627	19.4.4 锁定左边列	738
17.1 列表模型	628	19.5 表格选取	741
17.1.1 AbstractListModel	629	19.6 绘制和编辑	746
17.1.2 DefaultListModel	630	19.6.1 使用表格单元绘制器和编 辑器	746
17.2 列表选取	635	19.6.2 表格单元绘制器	752
17.3 列表单元绘制器	637	19.6.3 Default Table Cell Renderer 类	754
17.3.1 JList 属性	643	19.6.4 表格格式化绘制器	755
17.3.2 JList 事件	646	19.6.5 单元编辑器	756
17.3.3 JList 类总结	661	19.6.6 表格单元编辑器	757
17.3.4 AWT 兼容	665	19.6.7 实现 TableCellEditor 接口	759
17.4 本章回顾	666	19.7 表格行	767
第 18 章 组合框	667	19.7.1 行高	767
18.1 JComboBox 与 JList 的比较	667	19.7.2 绘制行	769
18.2 JComboBox 组件	667	19.8 表格装饰器	771
18.3 组合框模型	668	19.9 表格头部	779
18.3.1 ComboBoxModel	670	19.9.1 JTableHeader	779
18.3.2 MutableComboBoxModel	670	19.9.2 列头部绘制器和头部工具 提示	780
18.3.3 DefaultComboBoxModel	670	19.9.3 JTable 属性	785
18.4 组合框单元绘制器	671	19.9.4 表格事件	787
18.5 组合框键选取管理器	674	19.9.5 表格模型事件	788
18.5.1 使用缺省键选取管理器	675	19.9.6 TableColumnModel 事件	791
18.5.2 定制键选取管理器	677	19.9.7 列表选取事件	793
18.5.3 程序式的键选取	682	19.9.8 JTable 类总结	794
18.6 组合框编辑器	686	19.9.9 AWT 兼容	799
18.6.1 JComboBox 属性	696	19.10 本章回顾	799
18.6.2 JComboBox 事件	697	第 20 章 树	800
18.6.3 JComboBox 类总结	700	20.1 创建树	800
18.6.4 AWT 兼容	704	20.2 树节点	803
18.7 本章回顾	704	20.2.1 TreeNode 接口	803
第 19 章 表格	705	20.2.2 MutableTreeNode 接口	804
19.1 表格和滚动	705		
19.2 表格模型	707		
19.2.1 表格数据模型	708		

20.2.3 DefaultMutableTreeNode 类	804	22.1 JTextField	904
20.3 树路径	811	22.1.1 水平可视性和滚动偏移	907
20.4 树模型	814	22.1.2 布局单行文本域	910
20.5 树选取	822	22.1.3 使单行文本域有效	913
20.6 树单元绘制	827	22.1.4 JTextField 组件总结	916
20.6.1 DefaultTreeCellRenderer	827	22.1.5 JTextField 属性	917
20.6.2 Metal 界面样式	835	22.1.6 JTextField 事件	918
20.6.3 根节点和根句柄	835	22.1.7 JTextField 类总结	920
20.7 树单元编辑	836	22.1.8 AWT 兼容	923
20.7.1 扩展 DefaultCellEditor	837	22.2 JPasswordField	923
20.7.2 DefaultTreeCellEditor	838	22.2.1 JPasswordField 组件总结	924
20.8 绘制和编辑: 学习一个样例	842	22.2.2 JPasswordField 属性	924
20.8.1 Test 类	843	22.2.3 JPasswordField 类总结	925
20.8.2 SelectableFile 类和 FileNode 类	845	22.3 JTextArea	926
20.8.3 绘制器	846	22.3.1 JTextArea 组件总结	929
20.8.4 编辑器	848	22.3.2 JTextArea 属性	929
20.8.5 JTree 属性	851	22.3.3 JTextArea 类总结	930
20.8.6 树事件	852	22.3.4 AWT 兼容	932
20.8.7 JTree 类总结	863	22.4 JEditorPane	932
20.8.8 AWT 兼容	868	22.4.1 JEditorPane 属性	934
20.9 本章回顾	868	22.4.2 JEditorPane 事件	935
第 21 章 文本基础	869	22.4.3 JEditorPane 类总结	936
21.1 Swing 文本组件	869	22.5 JTextPane	939
21.2 动作	871	22.5.1 嵌入图标和组件	939
21.2.1 文本动作	871	22.5.2 用属性标记内容	941
21.2.2 动作和编辑工具包	875	22.5.3 JTextPane 属性	947
21.3 键映射	877	22.5.4 JTextPane 类总结	947
21.4 文档	880	22.6 AWT 兼容	949
21.4.1 定制文档	882	22.7 本章回顾	949
21.4.2 文档监听器	883	第 23 章 定制文本组件	950
21.5 加字符与加重器	887	23.1 概览	950
21.5.1 加字符	887	23.2 属性集和风格常量	952
21.5.2 加字符监听器	888	23.3 定制动作	954
21.5.3 定制加字符	889	23.4 视图	958
21.5.4 加重器	891	23.5 风格和风格的相关内容	962
21.6 撤销/恢复	893	23.6 元素	968
21.7 JTextComponent	897	23.7 本章回顾	971
21.8 本章回顾	903	第三部分 附录	972
第 22 章 文本组件	904	附录 A 类图	972
		附录 B 插入式界面样式常量	975

第一部分 Swing 基础

第 1 章 简介

Java 的基础类 (JFC) 是开发图形用户界面的 API 集。Java 的基础类包括以下 API:

- 抽象窗口工具包 (版本 1.1 及以后的版本)。
- 2D API。
- Swing 组件。
- 可访问性 API。

抽象窗口工具包 (Abstract Window Toolkit, AWT) 是 Java 开发用户界面最初的工具包。AWT 是建立 JFC 的主要基础,《Java 2 图形设计, 卷 I: AWT》对 AWT 有详细的介绍。

2D API 提供了 AWT 所缺乏的附加图形功能。例如, AWT 对图形操作只提供了一种规格的笔——即一个像素大小的正方形。而 2D API 除提供了不同大小的笔外, 还提供了丰富的二维着色能力。《Graphic Java 2, Volume IV: 2D API》一书中对 2D API 有详细的介绍。

Swing 是建立在 AWT 之上的[○]、包括大多数轻量组件的组件集。除提供了 AWT 所缺少的、大量的附加组件外, Swing 还提供了替代 AWT 重量组件的轻量组件。Swing 还包括了一个使人印象深刻的、用于实现包含插入式界面样式等特性的图形用户界面的下层构件。因此, 在不同的平台上, Swing 组件都能保持组件的界面样式特性, 如双缓冲、调试图形和文本编辑包等。

可访问性 (Accessibility) API 是一个类集, 其中的类使 Swing 组件能够与用于残疾用户的援助技术交互。JFC 还包括许多可访问性工具, 这些工具与可访问性 API 联合使用。

1.1 Swing 的历史

要了解 Swing, 首先必须了解 AWT, AWT 是 Swing 的基础。

Java 的发展速度超出了人们的想象, Java API 中最可视的部分——AWT 突然成为了人们关注的焦点。遗憾的是, 原来的 AWT 不能满足发展的需要。

原来的 AWT 不是为许多开发人员使用的、功能强大的用户界面 (UI) 工具包而设计的, 其设计目的是支持开发小应用程序中的简单用户界面。例如, 原来的 AWT 缺少许多面向对象 UI 工具包中所能见到的特性, 例如, 剪贴板、打印支持和键盘导航等特性在 AWT 中都不存在。原来的 AWT 甚至不包括弹出式菜单或滚动窗格等基本特性, 而弹出式菜单和滚动窗格是开发现代用户界面的两个基本元素。

此外, AWT 的下层构件还有严重的缺陷。人们使 AWT 适应基于继承的、具有很大伸缩性的事件模型。甚至更糟, 基于对等组件 (peer) 的体系结构也被用于 AWT, 该体系结构注定要成为 AWT 的致命弱点。

为了尽快推向市场和保持本地的界面样式, 于是产生了基于对等组件的体系结构, 而该体

[○] 参见 1.2 节“轻量组件与重量组件的比较”中对轻量组件与重量组件比较的讨论。

系结构注定是要失败的。对等组件是完成薄弱的 AWT 对象所委托任务的本地用户界面组件。对等组件负责完成所有的具体工作，包括绘制自己、对事件做出反应等，这使得 AWT 组件除了在适当的时间与其对等组件交互外无事可做。由于 AWT 类只是较复杂的本地对等组件的外壳，所以，AWT 的早期开发人员能在最快的时间^①内创建组件。例如，`java.awt.Panel` 类只包含十二行代码。

另外，对等组件的设计也有严重的缺点。首先，在大多数平台上，对等组件都是在本地窗口中绘制的。每个组件一个本地窗口实在不能得到高性能，为此，含有大量 AWT 组件的小应用程序付出了很高的性能代价。

把不同平台上的本地对等组件硬塞进 Java 框架中也是一个问题，使这些 AWT 组件跨平台的表现一致是完全不可能的。结果，不但没有实现急需的新组件，而且开发时间都浪费在修补对等组件的错误上和不兼容问题上了。

更糟的是，AWT 有很高的错误发生率。于是，第三方开始提供他们自己的工具包，这些工具包提供了更可靠的下层构件并提供了比 AWT 更多的功能。这些工具包之一是 Netscape 的 Internet 基础类 (IFC)，IFC 是一组建立在 NEXTSTEP 中的用户界面工具包概念基础上的一组轻量类。IFC 组件不是对等的，在许多方面胜过了 AWT 组件。IFC 还吸引了更多的开发人员加盟。

由于认识到 Java 领域很可能在标准用户界面工具包问题上出现分裂局面，JavaSoft 和 Netscape 达成了一个交易，共同实现 Java 基础类 (Apple 公司和 IBM 公司也参加了 JFC 的开发)。Netscape 开发人员与 Swing 工程师一起合作，以便把大部分的 IFC 的功能嵌入到 Swing 组件中。

起初打算让 Swing 类似于 Netscape 的 IFC。然而，随着时间的推移，在增加了插入式界面样式等特性并修改了设计之后，Swing 大大地偏离了它原来的目标。随着 Swing 1.1 版本的推出，虽然大量的 IFC 技术仍然嵌在 Swing 中，但是，Swing 与 IFC 相似的部分已大部分消失了。今天，在一个功能全面的用户界面工具包中，Swing 提供了 AWT 和 IFC 中最优秀的成份。

1.2 轻量组件与重量组件的比较

轻量组件首次出现在 AWT 1.1 版本中。AWT 最初只包括与本地对等组件相关联的重量组件，这些组件在它们自己的本地不透明窗口中绘制。

相反，轻量组件没有本地对等组件，而且在它们的重量容器的窗口中绘制。

由于轻量组件不在本地不透明的窗口中绘制，因此，它们可以有透明的背景。透明的背景使显示的轻量组件可以是非矩形的，虽然所有组件（重量的或轻量的）都有一个矩形边框。

Swing 组件几乎都是轻量组件，那些顶层容器：窗体、小应用程序、窗口和对话框除外。因为轻量组件是在其容器的窗口中绘制的，而不是在自己的窗口中绘制的，所以轻量组件最终必须包含在一个重量容器中。因此，Swing 的窗体、小应用程序、窗口和对话框都必须是重量组件，以便提供一个可以在其中绘制 Swing 轻量组件的窗口。

1.3 Swing 组件

Swing 包含 250 多个类，是组件和支持类的集合。Swing 提供了 40 多个组件，是 AWT 组件

^① 原来的 AWT 是在不足六个星期的时间内开发出来的。

的四倍。除提供替代 AWT 重量组件的轻量组件外, Swing 还提供了大量有助于开发图形用户界面的附加组件。

1.3.1 AWT 的替代组件

图 1-1 展示了用于替代 AWT 重量组件的 Swing 轻量组件。其中许多组件与它们所替代的 AWT 组件几乎是源代码兼容的。这使得替换 AWT 组件的工作相当简单。

除模仿 AWT 组件所提供的功能外, 几乎所有的 Swing 替代组件都有其他一些特性。例如, Swing 按钮和标签可显示图标和文本, 而 AWT 按钮和标签只能显示文本。

图 1-1 中所示的所有组件均使用 Windows 的界面样式。

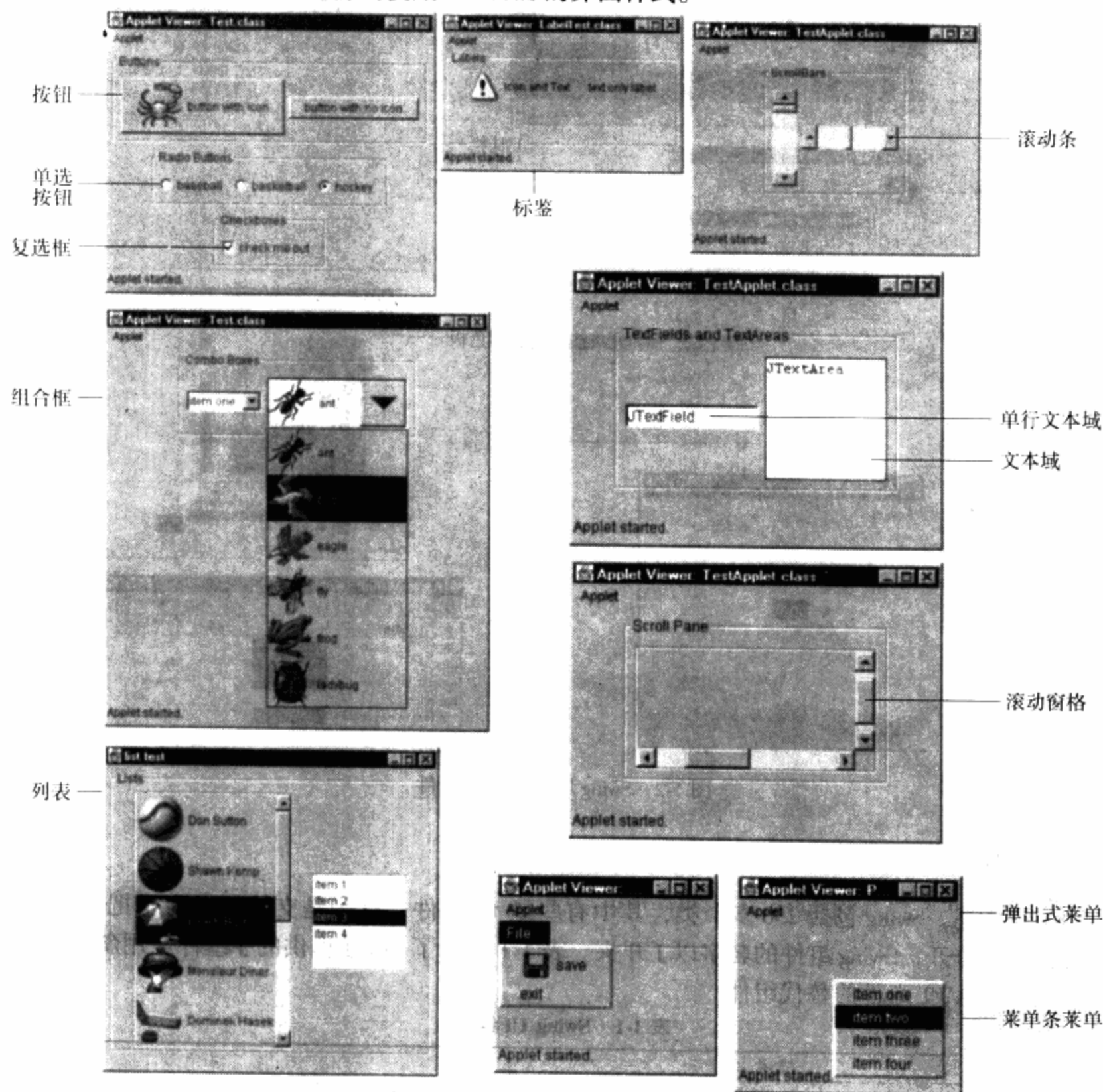


图 1-1 可替代 AWT 的 Swing 组件

1.3.2 Swing 增加的组件

除提供 AWT 重量组件的替代组件外, Swing 还提供了许多其他组件, 如表格、树、定制对

对话框等。图 1-2 示出了一些新的 Swing 组件。

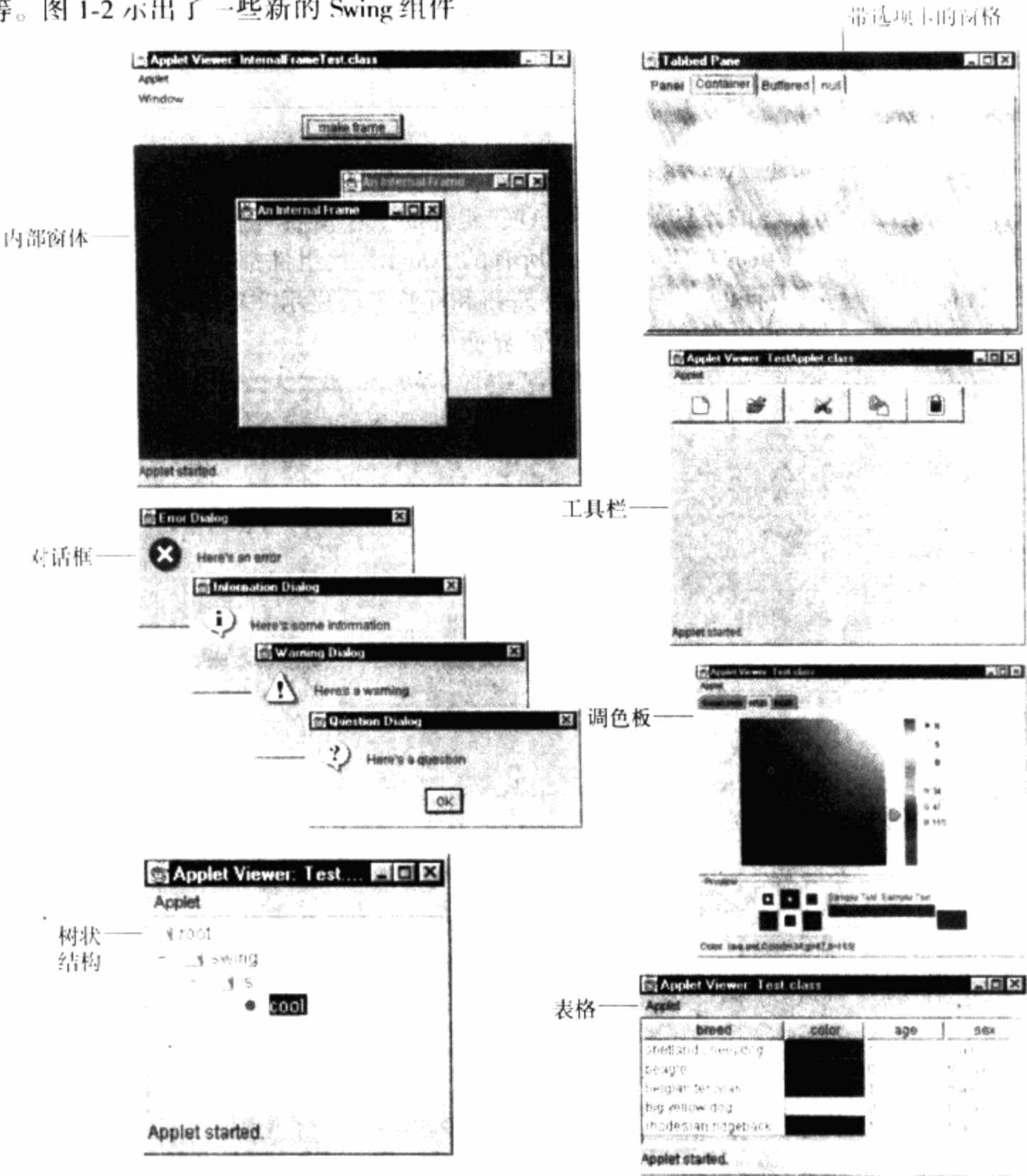


图 1-2 Swing 增加的一些组件

1.4 J 组件

如前所述，Swing 包括 250 多个类，其中有些是 UI 组件，有些是支持类。为了把 UI 组件和支持类区分开，Swing 组件的名字以 J 开头。表 1-1 列出了 Swing 提供的 J 组件。用斜体字表示的组件是 AWT 组件的替代组件。

表 1-1 Swing UI 组件

组件类	描述
JApplet	java.applet.Applet 类的扩展，它含有 JRootPane 的一个实例
JButton	能显示文本和图形的按钮，它是 AWT 按钮组件的替代组件
JCheckBox	能显示文本和图形的复选框，它是 AWT 选择组件的替代组件
JCheckBoxMenuItem	一个复选框菜单项，它是 AWT 的复选框菜单项组件的替代组件
JComboBox	带下拉列表的文本框，它是 AWT 选择组件的替代组件

(续)

组件类	描述
<i>JComponent</i>	所有轻量 J 组件的基类
<i>JDesktopPane</i>	内部窗体的容器
<i>JDialog</i>	Swing 对话框的基类, 它扩展了 AWT Dialog 类
<i>JEditorPane</i>	用于编辑文本的文本窗格
<i>JFrame</i>	扩展 java.awt.Frame 的外部窗体
<i>JInternalFrame</i>	在 JDesktopPane 中出现的内部窗体
<i>JLabel</i>	可显示文本和图形的标签, 它是 AWT 标签组件的替代组件
<i>JLayeredPane</i>	能够在不同层上显示组件的容器
<i>JList</i>	显示选项列表的组件, 它是 AWT 列表组件的替代组件
<i>JMenu</i>	菜单条中显示的一个菜单, 它是 AWT 菜单组件的替代组件
<i>JMenuBar</i>	用于显示菜单的菜单条, 它是 AWT 菜单条组件的替代组件
<i>JMenuItem</i>	菜单项, 它是 AWT 菜单项组件的替代组件
<i>JOptionPane</i>	显示标准的对话框, 如: 消息和问题对话框
<i>JPanel</i>	通用容器, 它是 AWT 面板和画布组件的替代组件
<i>JPasswordField</i>	<i>JTextField</i> 的扩展, 使输入的字符不可见
<i>JPopupMenu</i>	弹出式菜单, 它是 AWT 弹出式菜单组件的替代组件
<i>JProgressBar</i>	进度指示器
<i>JRadioButton</i>	单选按钮, 它是 AWT 复选框组件的替代组件
<i>JRootPane</i>	顶层容器, 它包含一个玻璃窗格、一个层窗格、一个内容窗格和一个可选的菜单条
<i>JScrollBar</i>	滚动条, 它是 AWT 滚动条组件的替代组件
<i>JScrollPane</i>	滚动窗格, 它是 AWT 滚动窗格组件的替代组件
<i>JSeparator</i>	水平或垂直分隔条
<i>JSlider</i>	滑杆
<i>JSplitPane</i>	有两个分隔区的容器, 这两个分隔区可以水平排列或垂直排列且分隔区的大小能自动调整
<i>JTabbedPane</i>	带选项卡的窗格
<i>JTable</i>	表格
<i>JTableHeader</i>	表格头
<i>JTextArea</i>	用于输入多行文本的文本域, 它是 AWT 文本域组件的替代组件
<i>JTextComponent</i>	文本组件的基类, 它替代 AWT 的 TextComponent 类
<i>JTextField</i>	单行文本域, 它替代 AWT 的单行文本域组件
<i>JTextPane</i>	简单的文本编辑器
<i>JToggleButton</i>	两种状态的按钮, 它是 JCheckBox 和 JRadioButton 组件的基类
<i>JToolBar</i>	工具条
<i>JToolTip</i>	当光标停留在一个组件上时, 该组件上显示的一行文字
<i>JTree</i>	用于按层次组织数据的结构控件
<i>JViewport</i>	用于浏览可滚动组件的视口
<i>JWindow</i>	外部窗口, 它是 java.awt.Window 的扩展

注: 斜体字表示的是 AWT 的替代组件

插入式界面样式

Swing 支持插入式界面样式, 界面样式的基础是“模型-视图-控制器”体系结构的变体。图1-3图解说明了在不同的界面样式下运行的小应用程序。

修改小应用程序或应用程序的界面样式不需要修改程序代码, 通过把 \$JDK_HOME/lib

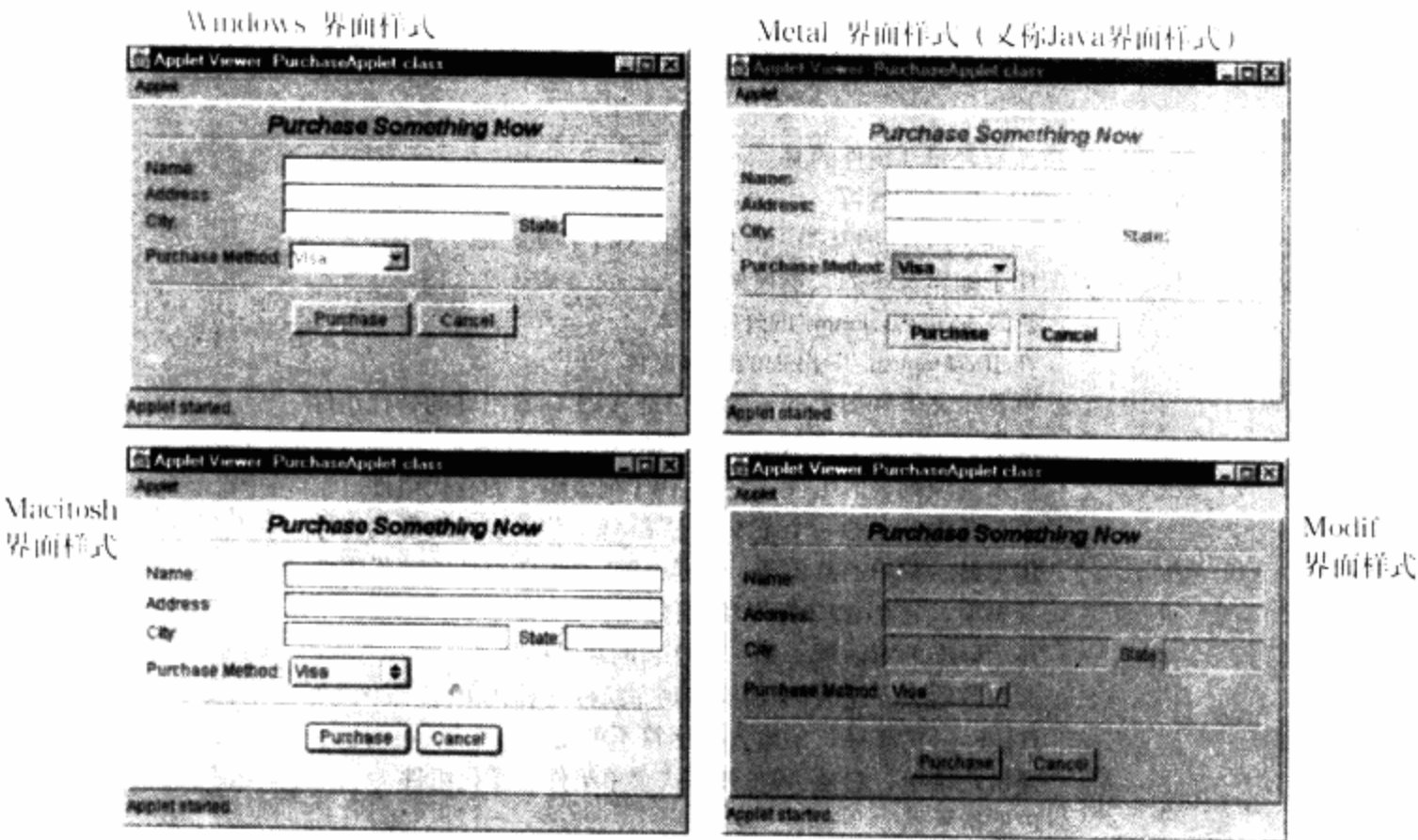


图 1-3 插入式界面样式

目录下的 `swing.properties` 文件中的 `swing.defaultlaf` 属性设置为所需的界面样式类型，就可在运行时刻设置缺省的界面样式。下面是 `swing.properties` 文件的一个例子，它通过指定 `defaultlaf` 属性来设置缺省的界面样式：

```
# swing.properties example file. Lines that begin with '#' are
# comments.
# The Mac look and feel is specified as the default look and
# feel below. If no look and feel is specified, then the default
# look and feel (metal) is used.
# the next line specifies which look and feels are installed.
swing.installedlafs = metal, motif, windows, mac

# default set to Mac look and feel
swing.defaultlaf = javax.swing.plaf.mac.MacLookAndFeel

# swing.defaultlaf = javax.swing.plaf.windows.WindowsLookAndFeel

# swing.defaultlaf = javax.swing.plaf.motif.MotifLookAndFeel
```

在第 7 章“插入式界面样式”中介绍了插入式界面样式的 Swing 实现。

1.5 Swing 包概览

Swing 由许多包组成，表 1-2 中列出了这些包。

表 1-2 Swing 包

包	描述
<code>com.sun.java.swing.plaf.motif</code>	用户界面代表类，它们实现 Motif 界面样式
<code>com.sun.java.swing.plaf.windows</code>	用户界面代表类，它们实现 Windows 界面样式
<code>javax.swing</code>	Swing 组件和实用工具
<code>javax.swing.border</code>	Swing 轻量组件的边框
<code>javax.swing.colorchooser</code>	JColorChooser 的支持类/接口
<code>javax.swing.event</code>	事件和侦听器类
<code>javax.swing.filechooser</code>	JFileChooser 的支持类/接口
<code>javax.swing.pending</code>	未完全实现的 Swing 组件

(读)

包	描述
<code>javax.swing.plaf</code>	抽象类，它定义 UI 代表的行为
<code>javax.swing.plaf.basic</code>	实现所有标准界面样式公共功能的基类
<code>javax.swing.plaf.metal</code>	用户界面代表类，它们实现 Metal 界面样式
<code>javax.swing.table</code>	JTable 组件的支持类
<code>javax.swing.text</code>	支持文档的显示和编辑
<code>javax.swing.text.html</code>	支持显示和编辑 HTML 文件
<code>javax.swing.text.html.parser</code>	html 文件的分析器类
<code>javax.swing.text.rtf</code>	支持显示和编辑 RTF 文件
<code>javax.swing.tree</code>	JTree 组件的支持类
<code>javax.swing.undo</code>	支持取消操作

swing 包是 Swing 提供的最大包，它包含将近 100 个类和 25 个接口。几乎所有的 Swing 组件都在 swing 包中，只有 JTableHeader 和 JTextComponent 是例外，它们分别在 swing.table 包和 swing.text 包中。

swing.border 包中含有数个在轻量 Swing 组件的边衬中画边框的类。border 包由一个 Border 接口、一个 AbstractBorder 类和 AbstractBorder 的许多具体扩展组成。

swing.event 包中定义了事件和事件监听器类，swing.event 包与 AWT 的 event 包类似。awt.event 和 swing.event 都包含事件类和监听器接口，它们分别响应由 AWT 组件和 Swing 组件激发的事件。例如，当在树组件中需要节点扩展（或折叠）的通知时，则要实现 Swing 的 TreeExpansionListener 接口，并把一个 TreeExpansionEvent 实例传送给 TreeExpansionListener 接口中定义的方法。TreeExpansionListener 和 TreeExpansionEvent 都是在 swing.event 包中定义的。

swing.pending 包包括仍没有完全实现的 Swing 组件。在 Swing1.1 FCS 中，pending 包包含选择器（日期、货币选择器）、计算器、弹出式按钮等等。pending 包中的组件最终会放到 swing 包中。

虽然 Swing 的表格组件（JTable）在 swing 包中，但它的支持类却在 swing.table 包中。表格模型、单元绘制器和编辑器等都在 swing.table 包中。

与 JTable 类一样，Swing 的树类 JTree（用于按层次组织数据的结构组件）也在 swing 包中，而它的支持类却在 swing.tree 包中。swing.tree 包提供树模型、树节点、树单元编辑器和绘制器等支持类。

Swing 有四个用于显示和编辑文档的包：`swing.text`、`swing.text.html`、`swing.text.html.parser` 和 `swing.text.rtf`。`swing.text` 包为 Swing 的文档模型提供了所有必须的下层构件，包括用于文档、元素、加字符、增亮、编辑工具包等的类和接口。`swing.text.html` 和 `swing.text.rtf` 包是 Swing 最小的两个包；它们分别提供用于实现 HTML（超文本标记语言）和 rtf（多文本格式）文档编辑器的编辑器工具包。`swing.text.html.parser` 包中含有分析 html 文件的支持类。

`swing.undo` 包为实现取消操作提供支持。

`swing.plaf` 包中的类形成了 Swing 插入式界面样式的 UI 代表部分的基础。UI 代表为它们的相关组件实现界面样式。

`swing.plaf` 包中的大多数类定义 UI 资源或扩展 `swing.ComponentUI` 类。`swing.ComponentUI` 类定义所有 UI 代表的公共行为。在 `swing.plaf` 包中的 UI 代表类（即名字以 UI 结尾的类）通常为

特定的组件定义附加的抽象方法。例如，`swing.plaf.ButtonUI` 类扩展 `swing.ComponentUI` 类并添加了抽象方法 `getDefaultMargin()`，该方法返回按钮边框与按钮内容之间的间距。

`swing.plaf.basic` 包扩展在 `swing.plaf` 包中定义的类，并且实现所有标准 Swing 界面样式共有的特性。例如，`swing.plaf.basic.BasicButtonUI` 类提供了 `swing.ComponentUI` 类和 `swing.plaf.ButtonUI` 类定义的一些方法的缺省实现。`BasicButtonUI` 类还为 `paint` 等方法提供了多个不同的实现（`paint` 方法绘制按钮的文本和图标）。`BasicButtonUI` 还以无操作形式实现其他与界面样式有关的方法，如 `paintButtonPressed()` 方法。这些方法将被特定的界面样式扩展所重载。

`metal` 和 `motif` 包实现相应界面样式的 UI 代表类。通常，UI 代表类扩展 `swing.plaf.basic` 包中的类。例如，`metal.ButtonUI` 类重载 `swing.plaf.basic.ButtonUI` 类中的 `paintButtonPressed` 方法的无操作实现。

`Swing.plaf.multi` 包支持界面样式复用。界面样式复用允许多个 UI 代表与一个组件相关联。例如，一个按钮 UI 代表可能同时与一个视觉 UI 代表和一个音频 UI 代表相关联，这样，当这个按钮被激活时，不仅能产生可视的反馈信息，还能播放一个声音。UI 复用的主要用途是使组件更具可访问性。

Swing 还提供了其他两种界面样式实现，即 `Mactintosh` 和 `Organic` 的界面样式。

1.6 Swing 与 AWT

对 Swing 最普遍的错误概念是认为其设计目的是用来替代 AWT 的。事实上，Swing 建立在 AWT 之上，如图 1-4 所示。

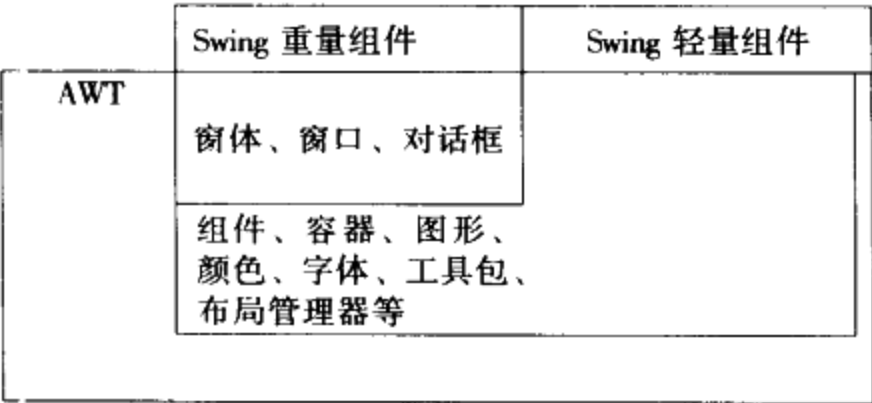


图 1-4 Swing 与 AWT 的关系

Swing 利用了 AWT 的下层构件，包括图形、颜色、字体、工具包和布局管理器；然而，Swing 没有使用 AWT 的组件。唯一与 Swing 有关的 AWT 组件是 `Frame`、`Window` 和 `Dialog` 类，它们分别被 Swing 的重量组件：`JFrame`、`JWindow` 和 `JDialog` 所扩展。Swing 使用 AWT 最好的部分来建立一个新的轻量组件集，并且丢弃了 AWT 中有问题的部分——重量组件。

Swing 是用来替代 AWT 的重量组件，而不是用来替代 AWT 本身。要了解 Swing，就必须具有 AWT 的基本知识。

Swing 除利用图形、字体、布局管理等 AWT 功能外，所有的 Swing 轻量组件基本上都是从 AWT 的 `Container` 类继承来的，而 AWT 的 `Container` 类又扩展了 AWT 的 `Component` 类。换句话说，Swing 不仅利用了 AWT 提供的下层构件，而且所有的 Swing 组件实际上都是 AWT 容器。注意：AWT `Container` 类本身是轻量的（即它没有对等组件[○]）且在其容器的窗口中绘制。

○ 实际上，`java.awt.Container` 有一个什么都不做的、待替换的对等组件。

对等组件与插入式界面样式的比较

Swing 组件和 AWT 组件都把与显示组件有关的许多工作和处理组件事件的工作交给其他对象来执行。对 AWT 组件而言，代表是一个本地对等组件，而对 Swing 组件而言，代表是 `ComponentUI` 类的一个扩展。虽然 Swing 组件和 AWT 组件都使用代表机制，但把工作交给其他对象处理所产生的结果在两个工具包中有明显的不同。

由于 AWT 组件把工作交给对等组件来完成，所以它们的行动很难扩展。例如，不可能把一幅图像添加到 AWT 的按钮上，这是因为按钮的绘制是由本地对等组件来完成的，而该对等组件可能是用 C++ 编写的，它的行为不能扩展。同样，因为文本域的对等组件负责增亮文本，所以，文本域增亮文本的方式也不能修改。要点是，任何由本地对等组件实现的行为是不能修改或扩展的。

另一方面，Swing 组件的代表（它的 `ComponentUI`）是 Swing 工具包中的一个 Java 类，它可以扩展以修改组件的行为。Swing 的插入式界面样式设计使用了改进的“模型-视图-控制器”体系结构，在这个体系结构中，组件的 UI 代表负责显示组件和处理输入事件的视图-控制器。当 Swing 组件配备了一个修改过的组件 UI 时，组件的可视外观或事件处理都是可以修改的。

1.7 开始学习

Swing 可与 JDK 1.1 或 1.2 版一起使用。JDK 1.2 版包含了 Swing，而 1.1 版却没有。要在 1.1 版中使用 Swing，必须从 JFC web 站点下载 Swing，其地址是：

<http://java.sun.com/products/jfc/index.html>。

在 Internet 浏览器中使用 Swing

在 Netscape Navigator 和 Internet Explorer 中都能用 Swing 小应用程序，但是，必须使用合适的浏览器版本，并确保包含了对 JDK 1.1 的支持。

1. Netscape Navigator

必须有 Netscape Navigator 4.04 或更新的版本，并且已安装了 JDK 1.1 的补丁程序。要下载 Netscape Navigator 和 JDK 1.1 补丁程序，请访问地址：

<http://developer.netscape.com/software/jdk/download.html>

图 1-5 示出了一个在 Netscape Navigator 中运行的 Swing 小应用程序。

在下载了一个合适的 Netscape Navigator 版本和 JDK 1.1 补丁程序之后，还必须确保 Netscape 可找到 Swing 的 jar 文件。使 Netscape 能找到 Swing 有两种方法：第一种方法是把 Swing 的 jar 文件拷贝到一个确定的 Netscape 目录中；第二种方法是修改系统的 CLASSPATH

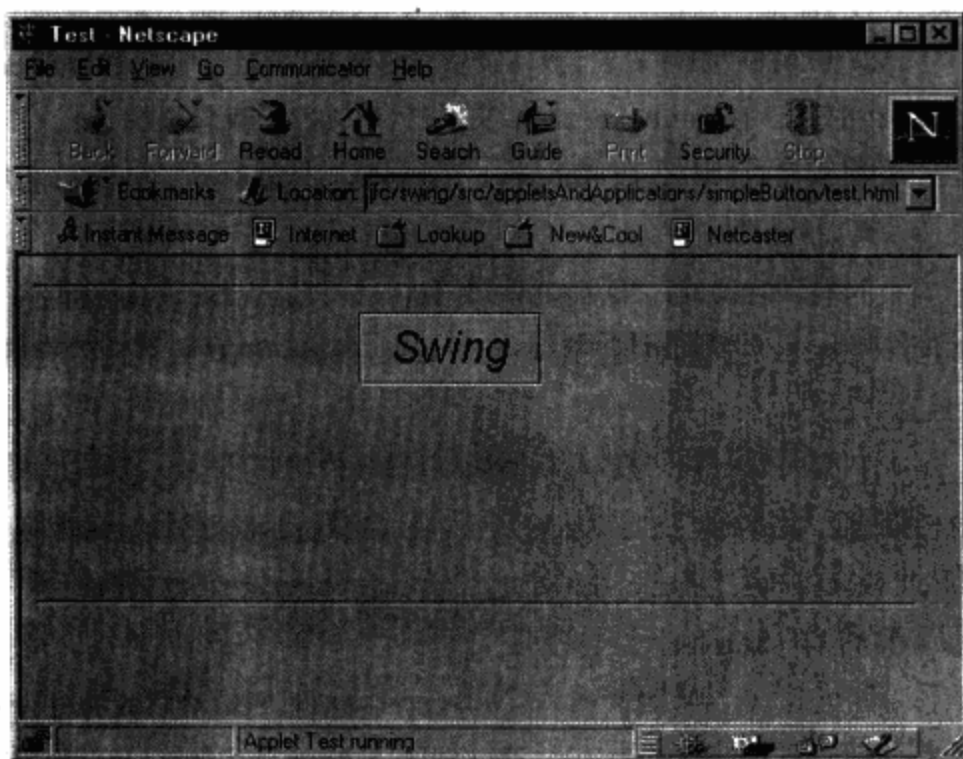


图 1-5 在 Netscape Navigator 中运行的 Swing 小应用程序

变量。本节介绍第一种方法，由于第二种方法对 Internet Explorer 和 Netscape Navigator 是相同的，所以我们将在“Internet Explorer”一节中介绍第二种方法。

可把 Swing 的 jar 文件拷贝到 Netscape 的 Java \ Classes 目录中。例如，如果把 Netscape 安装在 C 盘中，Swing 安装在 c: \ swing 目录下，则应把所有的 Swing jar 文件从 c: \ swing 拷贝到 c: \ programFiles \ Netscape \ Communicator \ Program \ Java \ Classes 下。只要有 Netscape Navigator 的最新版本，安装了 JDK 1.1 补丁程序，并且 Netscape 可找到 Swing 的 jar 文件，就具备了在 Netscape Navigator 中运行 Swing 小应用程序的条件。

2. Internet Explorer

Internet Explorer 的 4.0 版或更高版本支持 JDK 1.1。只要有合适的 Internet Explorer 版本并设置了系统的 CLASSPATH 变量，以便 Internet Explorer 能找到 Swing 的 jar 文件，就能在 Internet Explorer 中运行 Swing 小应用程序。下面介绍如何设置系统的 CLASSPATH 变量。

对 Windows NT 系统，进入 Windows 的“控制面板”，双击“系统”图标，在“系统属性”窗口中单击“Environment (环境)”选项卡，把 CLASSPATH 变量添加到“User Variables for Administrator”列表框中，如图 1-6 所示。

CLASSPATH 变量应该包括 JDK 的 classes.zip 文件和 swingall.jar 文件。例如，图 1-6 显示了 JDK 和 Swing 均安装在 D: \ 下时设置 CLASSPATH 变量的情况。添加(或修改)完 CLASSPATH 变量后，单击 OK 按钮关闭“System Properties”窗口，此时，需要重新启动系统，重启后，就可以在 Internet Explorer 中运行 Swing 小应用程序了。

对 Windows 95 系统，必须手工编辑 c: \ 下的 autoexec.bat 文件，只需在 autoexec.bat 文件中添加一项，如下所示：

```
SET CLASSPATH = C: \ jdk \ lib \ classes.zip; C: \ swing \ swingall.jar
```

同样，在 autoexec.bat 文件中添加(或修改)了 CLASSPATH 变量后，必须重新启动系统。

图 1-7 示出了在 Internet Explorer 中运行的 Swing 小应用程序。

3. Java 插件

当在 Netscape Navigator 或 Internet Explorer 中运行 Swing 小应用程序时，毫无疑问地还会发现许多错误。这些问题可能与 Swing 本身无关，可能是开发 Internet 上使用的 Java 小应用程序的主要缺点。

小应用程序开发人员面临的最困难的任务之一是使小应用程序在不同的浏览器中的表现是一致的。此外，由于历史原因，浏览器制造商已减慢了更新浏览器(与最新版 JDK 同步)的速度。幸运的是，针对这个问题 Sun 公司已推出了一个漂亮的解决方案，即它的 Java 插件(以前称作 Activator)。

通过把一个插件插入 Netscape 的 Netscape Navigator 中或在 Internet Explorer 中运行一个 Active X 控件来使用 Java 插件。插件或 Active X 控件有效地把 Sun 的 JDK 最新版本安装到浏览器中。使用 Java 插件保证了小应用程序在不同的浏览器中的一致性。

Java 插件唯一的缺点是要求对 HTML 文件做一些修改。当然，Sun 也提供了一个实用工具

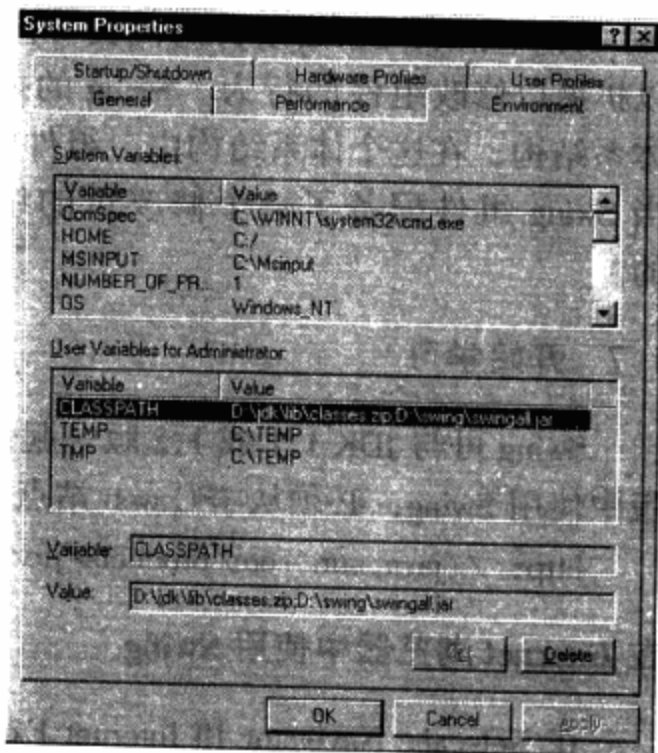


图 1-6 在 Windows NT 中配置 CLASSPATH 变量

来进行这种修改。要更多地了解免费获得 Java 插件的情况, 请访问下面的 Web 站点:

<http://java.sun.com/features/1998/04/plugin.html>

1.8 Swing 资源

除本书外, 还能找到许多学习 Swing 的资源。当需要解答对本书中没有介绍到的问题时, 就需要去寻找其他资源。

开始学习 Swing 的最好方法是学习 Swing 本身随带的例子代码。这些例子提供了许多小而完善的小应用程序和应用程序, 它们对 Swing 很多方面的特性做了练习。Swing 的这些例子可在 Swing 主目录的 examples 目录下找到。例如, 如果 Swing 安装在 c: \ swing 目录下, 则可以在 c: \ swing \ examples 目录下找到 Swing 的例子。

Internet 上也有许多 Swing 资源, 其中包括邮件列表和新闻组。下面列出的新闻组是解决 Swing 问题的好地方:

comp.lang.java.programmer

comp.lang.java.gui

此外, 还有许多邮件列表供喜爱 Swing 的初高级用户使用, 要了解邮件列表的有关信息, 请访问下面的 Web 站点:

<http://www.eos.dk/>

Swing Connection 是由 Sun 维护的 Swing 的正式站点。可在下面的 Web 站点中找到:

<Http://java.sun.com/products/jfc/tsc/>

1.9 本章回顾

Java 于 1995 年出现, 并迅速发展成为程序开发人员所喜爱的语言。Java 在重新定位于 Internet 和重命名为 Java 以前五年多的时间里, 它是以 Oak 语言的身份出现的 (Oak 是一种研究语言, Sun 公司打算使它成为 C++ 的一种更易于使用的和功能更强大的变体)。

虽然 Java 语言开发了许多年并且在 Sun 公司内部使用, 但是该语言没有用户界面工具包。当 Java 的优势开始显现时, 很明显, 它必须有用户界面工具包并应在最短的时间内开发出来。开发用户界面最快的方法是在本地组件 (又叫做对等组件) 上做大部分工作, 并在本地组件的顶层放一层 Java 类。这样, 在六个星期不到的时间内开发小组就实现了 AWT。

对等组件体系结构没有扩展性, 并导致了不同平台上不一致的问题产生。此外, AWT 没有可靠的面向对象基础; 例如, 最初的事件模型需要一个 switch 语句, 以便根据事件类型来决定激活哪段代码。这种 switch 语句是对面向对象的破坏; 这种根据对象类型进行切

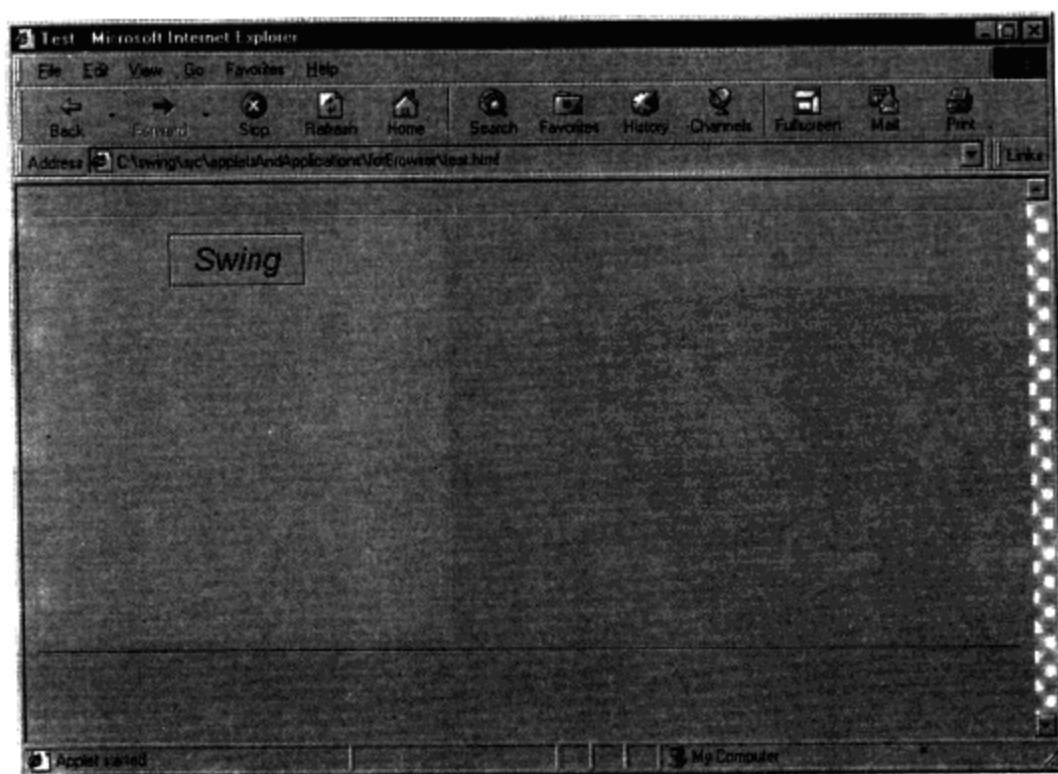


图 1-7 Internet Explorer 中运行的 Swing 小应用程序

换的 switch 语句应该通过多态性机制来处理^①。

Java 开发人员、Sun 公司或 Sun 公司的竞争者逐渐注意到原来的 AWT 的缺点。不久，出现了许多用来替代 AWT 的工具包。同时，Sun 发布了 AWT 的 1.1 版，它做了许多改进，包括一个新的事件模型和对轻量组件（非对等组件）的支持。然而，AWT 1.1 版还不够好用。

JavaSoft 认识到 Java 领域在用户界面工具包的使用上将会出现分裂，于是他们与 Netscape 合作开发 Swing 组件集。Netscape 和 Sun 公司的工程师用了将近一年半的时间来开发 Swing，Swing 在 AWT 上进行了巨大的改进。

虽然有些 Swing 组件替代了 AWT 的重量组件，但 Swing 不是 AWT 的替代品，而是 AWT 的扩展，Swing 使用了大量的 AWT 下层构件，包括对图形、字体和布局管理器的支持。要更深入地了解 Swing，就必须对 AWT 的下层构件有基本的了解^②。

与所有其他软件一样，Swing 还不完美。到现在为止，Swing 中仍有一些程序错误，在某些地方还表现出了一些设计缺陷，但它是一个可靠的用户界面工具包，比原来的 AWT 有了很大改进。

① switch 语句有时在面向对象设计中是有效的，但 AWT 的事件模型不是这种情况。

② 参见《Java2 图形设计，卷 1：AWT》。

第2章 Swing 的基本知识

本章介绍开发 Swing 小应用程序和应用程序时要用到的 Swing 的基本知识。

虽然 Swing 是 AWT 的扩展，但是两者的基本概念还是有许多不同之处。首先，Swing 小应用程序和应用程序的实现方式与 AWT 小应用程序和应用程序的实现方式有所不同。而且，如果开发人员想要开发同时使用 AWT 组件和 Swing 组件的小应用程序或应用程序，则还必须注意混合使用轻量组件和重量组件所带来的许多问题。

Swing 是线程不安全的，这就是说，在大多数情况下，只能从事件派发线程中访问 Swing 组件。本章将介绍采用这种方法的原因及使用这种方法所带来的结果，另外，本章还介绍了 Swing 提供的一些机制，这些机制使其他线程能从事件派发线程中执行代码。

2.1 小应用程序与应用程序

使用 Swing 组件的小应用程序和应用程序应该分别扩展 Swing 的 JApplet (java.applet.Applet 的一个扩展) 和 JFrame (java.awt.Frame 的一个扩展)。JApplet 和 JFrame 除具有它们的超类所提供的功能外，还提供对 Swing 的支持。虽然可以分别使用 Applet 类和 Frame 类来实现 Swing 的小应用程序和应用程序，但是，这样很可能出现事件处理问题和重新绘制问题。因此，应当总是使用 JApplet 和 JFrame 来实现 Swing 的小应用程序和应用程序。

JApplet 和 JFrame 都是只包含一个组件的容器，这个组件是 JRootPane 的一个实例，JRootPane 在 12.2 节“JRootPane”中介绍。目前，只需知道 JRootPane 包含一个称作内容窗格的容器即可。内容窗格包含与特定的小应用程序或应用程序有关的所有内容。这里，内容指包含在小应用程序和或应用程序中的组件。实际上，这就是说小应用程序和应用程序必须把组件添加到内容窗格中而不是把它们直接添加到小应用程序或应用程序（或根窗格）中。而且，我们不应该直接为 Swing 小应用程序或应用程序设置布局管理器。因为组件添加到内容窗格中，所以应该为内容窗格而不是小应用程序或应用程序设置布局管理器。

包含一个 JRootPane 实例的 Swing 容器重载用来添加组件和设置布局管理器的方法。这些方法会弹出提醒人们的异常信息：不能把组件直接添加到包含一个 JRootPane 实例的 Swing 容器中，也不能为该容器设置布局管理器。

2.1.1 小应用程序

图 2-1 所示的小应用程序包含一个 JLabel 实例，该实例有一个图标和一些文本。该小应用程序扩展 JApplet 并通过调用 JApplet.getContentPane() 方法来获得对其内容窗格的引用。这个标签随后被实例化并被添加到这个内容窗格中。

例 2-1 列出了图 2-1 所示的小应用程序的代码。

例 2-1 一个 Swing 小应用程序

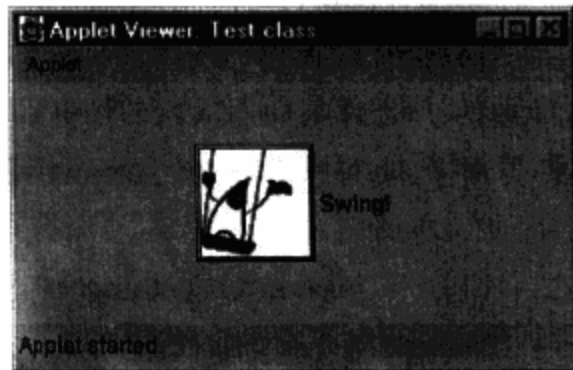


图 2-1 Swing 小应用程序

```
import javax.swing.* ;  
import java.awt.* ;
```



```

import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();

        Icon icon = new ImageIcon (" swing.gif",
                                   " An animated GIF of Duke on a swing");

        JLabel label = new JLabel (" Swing!", icon,
                                   SwingConstants.CENTER);

        contentPane.add (label, BorderLayout.CENTER);
    }
}

```

JApplet 类使用 BorderLayout 的一个实例作为其内容窗格的布局管理器。为了强调这一点，例 2-1 的小应用程序指定其布局约束条件为 BorderLayout.CENTER，它使标签在内容窗格中居中显示。用 BorderLayout 来布局组件的缺省约束条件是 BorderLayout.CENTER，所以，在该小应用程序中指定这个布局约束条件不是必须的。

注意 当在 Internet Explorer 中使用例 2-1 的小应用程序时，必须使用 BorderConstraints.CENTER。

JApplet 的内容窗格用 BorderLayout 的一个实例来布局组件。记住这个点是很重要的，因为 java.applet.Applet 与 JApplet 不同，它使用 FlowLayout 的一个实例来布局组件。

2.1.2 JApplet 类

Swing 的 JApplet 类扩展 java.applet.Applet 并实现 Accessibility 接口和 RootPaneContainer 接口。Accessibility 接口是可访问包的一部分，而 RootPaneContainer 接口（如其名字所指出的）是一个包含根窗格的容器。RootPaneContainer 接口被所有包含一个 JRootPane 实例的 Swing 容器所实现。

类总结 2-1 中列出了 JApplet 提供的 public 和 protected 方法。

类总结 2-1 JApplet

扩展：java.applet.Applet

实现：javax.accessibility.Accessible、RootPaneContainer

1. 构造方法

public JApplet ()

JApplet 只提供了一个不带参数的构造方法。由于小应用程序是由浏览器（或小应用程序阅读器）进行实例化的，所以，正常情况下，不需要直接把 JApplet 的一个实例进行实例化。要了解直接实例化一个 JApplet 实例的情况，请参见 2.1.5 节“小应用程序/应用程序组合”。

2. 方法

(1) 从 java.awt.Container 中重载而获得的方法

protected void addImpl (Component, Object, int)

public void setLayout (LayoutManager)

public void addNotify ()

public void removeNotify ()

上面列出的四种方法都是重载 java.awt.Container 类中的方法而得到。



AddImpl () 是最终把组件添加到容器中的方法。如果直接把组件添加到小应用程序中, 那么 JApplet.addImpl () 将弹出一个异常信息。这个异常中所显示的消息是定制的^①。例如, 如果例 2-1 小应用程序中的标签直接添加到该小应用程序中, 那么异常信息将如下显示:

```
Java.lang. Error: Do not use Test.add () use Test.getContentPane ().add () instead
    at javax.swing.JApplet.createRootPaneException (JApplet.java: 198)
    at javax.swing.JApplet.addImpl (JApplet.java: 220)
    at java.awt.Container.add (Container.java: 179)
    at Test.init (Test.java: 11)
```

与 JApplet 重载 addImpl () 的原因一样, JApplet 也重载 setLayout ()。如果设置了小应用程序的布局管理器, setLayout () 将会弹出一个异常信息。如果修改例 2-1 的小应用程序, 让该小应用程序试图设置它的布局管理器, 则将弹出带有下面错误消息的异常信息:

```
Java.lang. Error: Do not use Test.setLayout () use Test.getContentPane ().setLayout () instead
    at javax.swing.JApplet.createRootPaneException (JApplet.java: 198)
    at javax.swing.JApplet.setLayout (JApplet.java: 244)
    at Test.init (Test.java: 10)
    at sun.applet.AppletPanel.run (AppletPanel.java: 287)
    at java.lang.Thread.run (Thread.java: 474)
```

当实例化一个组件的对等组件时, 将调用 addNotify () 方法。JApplet 重载 addNotify () 以激发键盘事件并把小应用程序的可见性设置为 true。

(2) 根窗格/内容窗格/玻璃窗格

```
protected JRootPane createRootPane ()
protected boolean isRootPaneCheckingEnabled ()
protected void setRootPaneCheckingEnabled (boolean)

public Container getContentPane ()
public Component getGlassPane ()
public JLayeredPane getLayeredPane ()
public JRootPane getRootPane ()
public void setContentPane (Container)
public void setGlassPane (Component)
public void setLayeredPane (JLayeredPane)
public void setRootPane (JRootPane)
```

Swing 小应用程序通过调用 protected JApplet.createRootPane 方法, 接着, 这个方法又调用 setRootPane () 方法来创建根窗格。createRootPane 方法可以被 JApplet 的扩展所重载, 以便替代 JRootPane 类的扩展作为该小应用程序的根窗格。

如前所述, 把组件直接添加到 JApplet 的一个实例中或显式地设置其布局管理器都可能会信息弹出一个异常。然而, 有时必须把 JRootPane 的一个实例直接添加到小应用程序中, 并且不能弹出异常信息。为此, 可以指定当允许根窗格检查 (root pane checking) 时, 才弹出异常信息。通过调用以 boolean 值为参数的 setRootPaneCheckingEnabled () 方法来设置一个标志, 该标志跟踪是否允许根窗格检查。如果这个 boolean 值是 true, 则说明允许根窗格检查, 如果这个 boolean 值是 false, 则说明禁止根窗格检查。

isRootPaneCheckingEnabled () 方法返回最后传送给 setRootPaneCheckingEnabled () 方法的 boolean 值。

① 消息是通过 JApplet 的扩展的名字定制的。

注意，`setRootPaneCheckingEnabled()` 和 `isRootPaneCheckingEnabled()` 都是 `protected` 方法。虽然不可能把组件直接添加到 `JApplet` 的一个实例中或显式地设置其布局管理器，但是，实现可以控制是否允许根窗格检查的 `JApplet` 的扩展是可能的。这种功能使 `JApplet` 的扩展能够在需要时直接添加组件或设置小应用程序的布局管理器。

实际中，很少重载 `JApplet.createRootPane()`，`JApplet` 的扩展也很少用 `setRootPaneCheckingEnabled()` 来直接添加组件或设置小应用程序的布局管理器。

上面列出的第二组方法是由 `RootPaneContainer` 接口定义的。这些方法能够获取和设置包含在 `JRootPane` 的一个实例中的容器。`JRootPane` 和 `RootPaneContainer` 将在第 12 章 12.2 节“`JRootPane`”中介绍。

(3) 可访问的相关内容/菜单栏/键盘事件/更新

```
public AccessibleContext getAccessibleContext()
public JMenuBar getJMenuBar()
public void setJMenuBar(JMenuBar)
protected void processKeyEvent(KeyEvent)
public void update(Graphics)
```

`getAccessibleContext()` 返回 `AccessibleContext` 的一个实例，这个实例把小应用程序的可访问信息提供给可访问工具。

`JApplet` 的实例可以有一个菜单栏，它是由 `setJMenuBar` 方法指定的。注意，Swing 小应用程序能有一个菜单栏，而 AWT 小应用程序却不能。参见图 2-2。

实际上有两种把菜单栏添加到 Swing 小应用程序中的方法。一种方法当然是调用 `JApplet.setJMenuBar`，另一种方法是获得对小应用程序根窗格的引用，然后把菜单栏直接添加到根窗格中。

重载 `ProcessKeyEvent()` 来处理键绑定问题。有关 Swing 组件中键击处理的更多信息，请参见 4.8 节“键击处理”。

重载 `JApplet.update` 方法以便直接调用 `paint()`。缺省时，AWT 组件将实现它们的 `update` 方法以便擦除背景，然后调用 `paint()`。这种技术在组件反复更新时，会导致许多闪烁。有关绘制和更新 AWT 组件的更多信息，请参见《Java 2 图形设计，卷 I：AWT》。

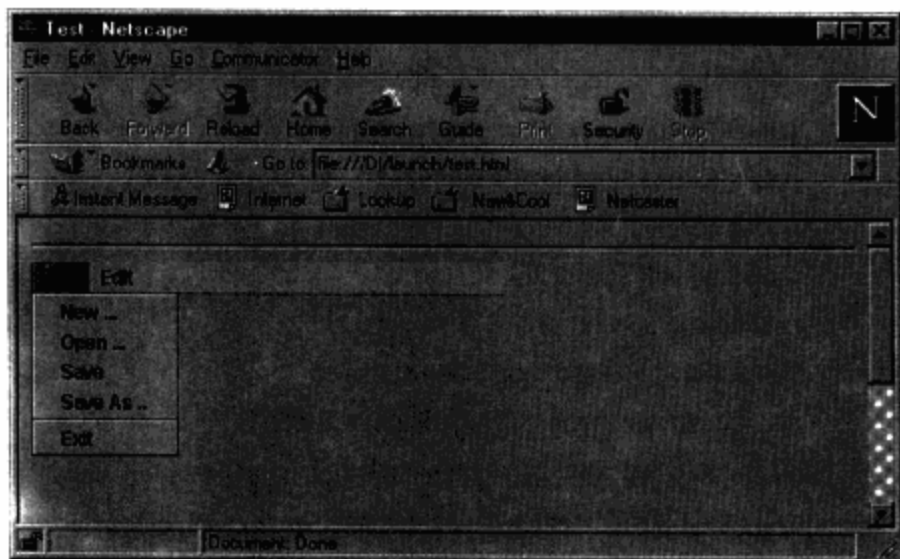


图 2-2 一个带菜单栏的 Swing 小应用程序

Swing 提示

JApplet 和 JFrame 的内容窗格使用一个 BorderLayout 实例

如果你用 AWT 开发过应用程序，就一定熟悉这样一个事实：`java.applet.Applet` 使用一个 `FlowLayout` 实例作为其布局管理器，而 `java.awt.Frame` 则使用一个 `BorderLayout` 实例作为其布局管理器。

由于 AWT 小应用程序和应用程序使用不同的布局管理器，所以，当把小应用程序移植为应用程序时或把应用程序移植为小应用程序时，就可能造成混乱，这里还没有涉及到实现一个

小应用程序和应用程序组合的情况。相比之下, Swing 在小应用程序和应用程序的内容窗格中使用相同的布局管理器 (即一个 BorderLayout 实例)。

2.1.3 应用程序

例 2-2 所示的应用程序与例 2-1 所示的小应用程序在功能上是完全相同的。它们都把 JLabel 的一个实例添加到它们的根窗格的内容窗格中。

例 2-2 列出了图 2-3 所用的应用程序的代码。

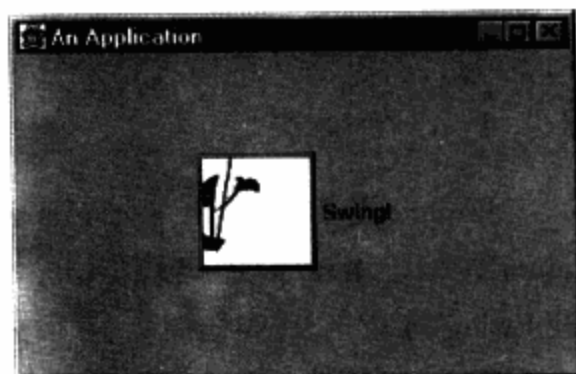


图 2-3 一个 Swing 应用程序

例 2-2 一个 Swing 应用程序

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    public Test () {
        super (" An Application");

        Container contentPane = getContentPane ();

        Icon icon = new ImageIcon (" swing.gif",
                                   " An animated GIF of Duke on a swing");

        JLabel label = new JLabel (" Swing!", icon,
                                   SwingConstants.CENTER);

        contentPane.add (label, BorderLayout.CENTER);
    }

    public static void main (String args []) {
        JFrame f = new Test ();

        f.setBounds (100, 100, 300, 250);
        f.setVisible (true);
        f.setDefaultCloseOperation (DISPOSE_ON_CLOSE);

        f.addWindowListener (new WindowAdapter () {
            public void windowClosed (WindowEvent e) {
                System.exit (0);
            }
        });
    }
}
```

应用程序比小应用程序要稍微复杂些, 这是因为它们不是在浏览器内部运行的, 即浏览器不启动它们也不设置它们的大小。应用程序必须提供 main 方法, 必须把一个窗体实例化, 随后确定该窗体的大小[⊖] 并使该窗体可见。

例 2-2 中的应用程序还设置窗体的缺省关闭操作并添加一个窗口监听器, 该监听器在窗体被关闭后会退出这个应用程序。有关 Swing 窗体的缺省关闭操作的更多信息, 请参见 2.1.4 节“JFrame 类”。

Swing 小应用程序和应用程序有许多共同点。它们都含有一个 JRootPane 实例, 都必须把组

⊖ 可使用 JFrame.pack ()显式地给出窗体的大小。

件添加到根窗格的内容窗格中。而且，不能显式地设置 Swing 小应用程序或 Swing 应用程序的布局管理器。

2.1.4 JFrame 类

JFrame 类扩展 `java.awt.Frame`，与 JApplet 类似，它也实现 Accessible 接口和 RootPaneContainer 接口。JFrame 还实现 `Swing.WindowsConstants` 接口，该接口定义缺省关闭操作的常量。有关 Swing 常量的更多信息，请参见 6.4 节“Swing 常量”。

JFrame 实现许多在 JApplet 中能找到的、相同的方法。与 JApplet 类似，为了不显式地设置其布局管理器或不把组件直接添加到窗体中，JFrame 重载 `setLayout` 和 `addImpl` 方法。JFrame 实现了所有在 RootPaneContainer 接口中定义的方法，还实现了允许和禁止根窗格检查的方法。JFrame 还实现了确定当前是否启用了根窗格检查的方法。

类总结 2-2 总结了 JFrame 类。

类总结 2-2 JFrame

扩展：`java.applet.Frame`

实现：`javax.accessibility.Accessible`、`RootPaneContainer`

1. 构造方法

`public JFrame ()`

`public JFrame (String title)`

JFrame 有两个构造方法，一个构造方法不带参数，一个构造方法以一个字符串为参数，该字符串代表窗体的标题。

浏览器或小应用程序阅读器会调用 Swing 小应用程序的构造方法，因此，通常不需要开发人员编写代码来调用它的构造方法，但是，应用程序必须负责构造窗体并负责设置窗体的大小。通常为 JFrame 的实例选择带一个字符串的构造方法，不带参数的构造方法将产生没有标题的窗体。

(1) 与 JApplet 交叠的方法

`protected void addImpl (Component, Object, int)`

`protected JRootPane createRootPane ()`

`public AccessibleContext getAccessibleContext ()`

`public Container getContentPane ()`

`public Component getGlassPane ()`

`public JMenuBar getJMenuBar ()`

`public JLayeredPane getLayeredPane ()`

`public JRootPane getRootPane ()`

`protected boolean isRootPaneCheckingEnabled ()`

`protected void processKeyEvent (KeyEvent)`

`public void setContentPane (Container)`

`public void setGlassPane (Component)`

`public void setJMenuBar (JMenuBar)`

`public void setLayeredPane (JLayeredPane)`

`public void setLayout (LayoutManager)`

`protected void setRootPane (JRootPane)`

`protected void setRootPaneCheckingEnabled (boolean)`

`public void update (Graphics)`

上面列出的 JFrame 方法与 JApplet 中定义的方法交叠。其中的大部分方法与 JApplet 中相应方法的实现方式是相同的。例如，如果允许根窗格检查，则 JFrame.setLayout 和 JFrame.addImpl 都将弹出一个异常信息。

有关上述方法的更多信息，请参见“类总结 2-1JApplet”。

(2) 窗体初始化/缺省的关闭操作/ 窗口事件

```
protected void frameInit ()
public int getDefaultCloseOperation ()
public void setDefaultCloseOperation (int)
protected void processWindowEvent (WindowEvent)
```

JFrame 构造方法调用 frameInit 方法来初始化窗体。JFrame 的 frameInit () 方法允许窗体的键盘事件和窗口事件，设置窗体的根窗格和背景色，并允许根窗格检查。如果缺省的设置不令人满意的话，也可扩展 JFrame 以重载 frameInit ()。

使用 AWT 窗体时，开发人员要负责处理窗口关闭事件。通常，这需要重载事件处理方法，需要简单地隐藏窗口或隐藏窗口并清除其本地资源。而 Swing 通过把一个缺省关闭操作与每一个 JFrame 实例相关联来使窗口的关闭事件较容易处理。可以用 setDefaultCloseOperation 方法来设置缺省的关闭操作，而且可以用 getDefaultCloseOperation () 来获取缺省的关闭操作。可以传送给 setDefaultCloseOperation () 的 integer 值在 WindowConstants 类中定义，表 2-1 列出了 integer 值。

表 2-1 WindowConstants public 常数

方法名	实现
DO _ NOTHING _ ON _ CLOSE	关闭窗口时什么也不做
HIDE _ ON _ CLOSE	关闭窗口时隐藏该窗口
DISPOSE _ ON _ CLOSE	关闭窗口时隐藏该窗口并清除其本地资源

如果没有显式地设置 JFrame 的缺省关闭操作，则缺省值是 DO _ NOTHING _ ON _ CLOSE。

DISPOSE _ ON _ CLOSE 隐藏窗体并清除与这个窗体有关的系统资源。如果该窗体是应用程序窗体，则在该窗体清除后，应用程序将继续运行。例如，例 2-2 所列的应用程序把应用程序窗体的缺省关闭操作设置为 DISPOSE _ ON _ CLOSE，但是，应用程序仍然负责处理窗体关闭事件。到应用程序得到窗体已关闭（当调用 windowClosed 方法时）窗体已隐藏并清除的通知时，应用程序仍在运行；结果，应用程序在 windowClosed 方法中调用 System.exit ()。

2.1.5 小应用程序/应用程序的组合

有时需要实现这样一个源文件，它既可作为应用程序运行又可作为小应用程序运行。例 2-3 示出了一种实现小应用程序/应用程序组合的方法。

例 2-3 Swing 小应用程序/应用程序组合

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
```

```

        Icon icon = new ImageIcon ("swing.gif",
                                   "An animated GIF of Duke on a swing");

        JLabel label = new JLabel ("Swing!", icon,
                                   SwingConstants.CENTER);

        contentPane.add (label);
    }

    public static void main (String args []) {
        // f is final because it is accessed from
        // the inner class below
        final JFrame f = new JFrame ();
        JApplet applet = new Test ();

        applet.init ();

        f.setContentPane (applet.getContentPane ());
        f.setBounds (100, 100, 308, 199);
        f.setTitle (" An Application");
        f.setVisible (true);

        f.setDefaultCloseOperation (
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener (new WindowAdapter () {
            public void windowClosed (WindowEvent e) {
                System.exit (0);
            }
        });
    }
}

```

其思想是实现一个小应用程序，这个小应用程序包含一个 `main` 方法。这个 `main` 方法把 `JFrame` 实例化，而且还创建这个小应用程序的一个实例。在调用小应用程序的 `init` 方法后，窗体用该小应用程序的内容窗格来替代该窗体的内容窗格。这个窗体接着设置其边界和标题，并把它的可见性设置为 `true`。

从本质上讲，这种技术会产生共享一个内容窗格的应用程序和小应用程序。当例 2-3 中的代码被编译后，它可以既作为小应用程序运行又可以作为应用程序运行。

应该注意的是，作为应用程序/小应用程序组合实现的应用程序，在使用 `main` 方法创建的小应用程序实例时必须非常小心。因为浏览器或小应用程序阅读器不能把这种小应用程序实例化，所以这种小应用程序是不完善的（从技术上说，它没有小应用程序的相关内容）。因此，这种小应用程序不能使用，例如，用 `Applet.getImage` 方法来获取一幅图像。实际应用中，也没有那么多限制，因为应用程序除借用小应用程序的内容窗格外不需要使用小应用程序。例如，应用程序通常使用 AWT 工具包来获取图像，因此，不需要使用 `Applet.getImage` 方法。

Swing 提示

不要直接把组件添加到 Swing 小应用程序或应用程序中，也不要显式地设置其布局管理器

Swing 小应用程序和应用程序都有一个 `JRootPane` 实例，该实例又含有一个称作内容窗格的容器。小应用程序或应用程序的内容（即组件）必须添加到内容窗格中。如果把组件直接添加到 `JApplet` 或 `JFrame` 的实例中，则会弹出一个异常信息，指出只能把组件添加到内容窗格中。

Swing 小应用程序和应用程序都使用 BorderLayout 布局管理器来布局它们的 JRootPane 实例，并且不允许显式地设置它们的布局管理器。如果试图显式地设置 JApplet 或 JFrame 的布局管理器，则会弹出一个异常信息，指出不可以显式地设置其布局管理器。

2.2 GJApp

本书介绍的应用程序都是在 GJApp 类的帮助下实现的，该类提供了一个状态区，并能从属性文件中读取资源。图 2-4 所示的应用程序是一个 JFrame 扩展，这个扩展用 GJApp 类来访问一个状态区，这个状态区显示从 GJApp.properties 文件中获得的一个字符串。

GJApp.properties 文件定义了一个属性：

```
# Simple properties file
statusAreaText = text in the status area
```

例 2-4 列出了图 2-4 所示的应用程序的代码。

例 2-4 使用 GJApp 类

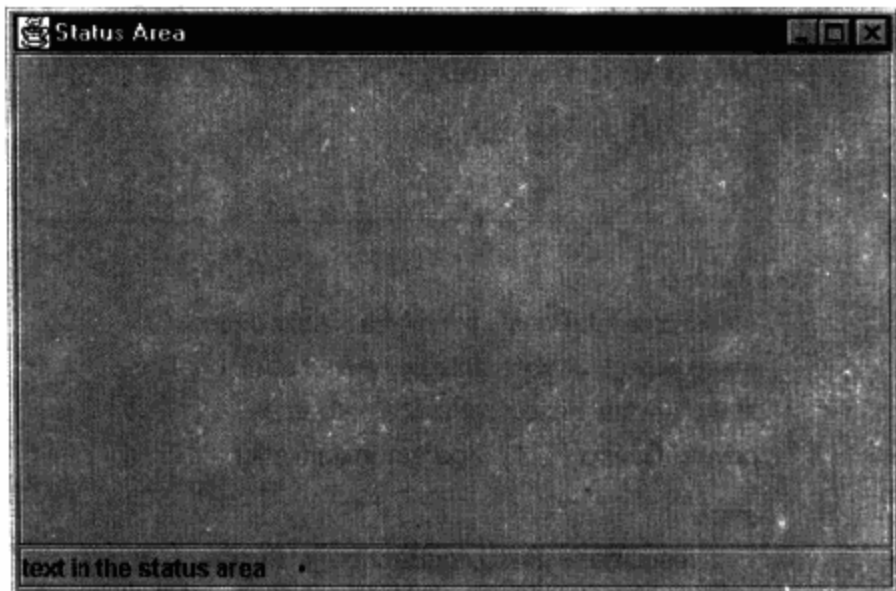


图 2-4 GJApp: 状态区和资源

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JFrame {
    public Test () {
        Container contentPane = getContentPane ();
        JPanel panel = new JPanel ();

        panel.setBorder (BorderFactory.createEtchedBorder ());
        contentPane.add (panel, BorderLayout.CENTER);

        //Add GJApp's status area to contentPane
        contentPane.add (GJApp.getStatusArea (), BorderLayout.SOUTH);

        // Update GJApp's status area with resource string
        GJApp.showStatus (GJApp.getResource ("statusAreaText"));
    }

    public static void main (String args []) {
        // launch application
        GJApp.launch (new Test (), "Status Area", 300, 300, 450, 300);
    }
}
```

这个应用程序创建 JPanel 的一个实例，指定该实例为内容窗格的中心组件。用面板来突出状态区上面的空间，并且这个面板还有一个蚀刻边框。

应用程序通过调用 static GJApp.getStatusArea 方法来获取对 GJApp 状态区的引用。状态区指定为内容窗格南边的组件。

static GJApp.showStatus 方法以 statusAreaText 资源的字符串为参数把这个状态区初始化。资

源的字符串用 `static GJApp.getResource` 方法来获得。

GJApp 类有三个功能：

- 初始化并显示传送给 `static launch` 方法的窗体。
- 提供对小应用程序状态区面板的访问。
- 从 `GJApp.properties` 文件中查找资源字符串。

例 2-5 列出了 GJApp 类。

例 2-5 GJApp 类

```
class GJApp {
    static private JPanel statusArea = new JPanel ()
    static private JLabel status = new JLabel ("");
    static private ResourceBundle resources;
    private GJApp () {} //defeat instantiation

    static {
        resources = ResourceBundle.getBundle (
            " GJApp", Locale.getDefault ());
    };

    public static void launch (final JFrame f, String title,
                               final int x, final int y,
                               final int w, int h) {
        f.setTitle (title);
        f.setBounds (x, y, w, h);
        f.setVisible (true);

        statusArea.setBorder (BorderFactory.createEtchedBorder ());
        statusArea.setLayout (new FlowLayout (FlowLayout.LEFT, 0, 0));
        statusArea.add (status);
        status.setHorizontalAlignment (JLabel.LEFT);

        f.setDefaultCloseOperation (
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener (new WindowAdapter () {
            public void windowClosed (WindowEvent e) {
                System.exit (0);
            }
        });
    }

    static public JPanel getStatusArea () {
        return statusArea;
    }

    static public void showStatus (String s) {
        status.setText (s);
    }

    static String getResource (String key) {
        return (resources == null)? null:
            resources.getString (key);
    }
}
```

严格地说，GJApp 是一个帮助类，它实现独有的 `static` 方法。GJApp 的实例不能实例化，这是 GJApp `private` 构造方法强加的规定。

一个 static 代码块（它在 main（）方法之前执行）试图获得对 GJApp.properties 文件资源包的一个引用。在 GJApp.getResource 方法中使用这个资源包可以获得一个与一个给定资源关键字相关的字符串。

GJApp.launch 方法为传送给它的窗体设置边界和标题，把这个窗体的可见性设置为 true，并打开这个窗体。这个 launch 方法还配置状态区并把窗体的缺省关闭操作设置为 WindowConstants.DISPOSE_ON_CLOSE。添加到这个窗体中的窗口监听器在窗口关闭时会退出该应用程序。

GJApp 类用 getStatusArea 方法访问其状态区面板。与小应用程序一样，GJApp 类用 showStatus 方法来更新状态区。

注意 本书以后介绍的应用程序都是在 GJApp 类帮助下实现的。但是，为了简短些，例 2-5 是本书中唯一列出了 GJApp 类的地方。

2.3 混合使用 Swing 组件和 AWT 组件

原来的 AWT 只是为重量组件设计的；在 AWT 1.1 版本发布前，还没有轻量组件。结果，AWT 不得不重做 AWT，以提供轻量组件。

任何软件开发人员都可证实，把一个复杂的系统和以前未预见的设计组合起来不是一个简单的任务，把轻量组件合并到 AWT 中也不例外。直到现在，在一个小应用程序或应用程序中混用轻量组件和重量组件还是有许多问题，尤其是把重量组件嵌入轻量容器中时更是如此。

2.3.1 层序

组件的层序是指同一容器中组件之间显示的层次关系。

如果容器是同类的（即它包含的组件都是轻量组件或都是重量组件），则按组件被添加到容器中的顺序来确定其层序。第一个被添加到容器中的组件有最高的层序，即它在同一容器中所有其他组件的上面显示。最后添加到容器中的组件的层序最低，即它在同一个容器中的所有其他组件的下面显示。

如果容器是异类的（即它既有轻量组件又有重量组件），则事情要稍微复杂些。从第 1.2 节“轻量组件与重量组件的比较”中，我们知道，轻量组件不是显示在它们自己的窗口中，而是显示在它们的重量容器的窗口中。所以，轻量组件的层序与重量容器的层序相同。如果多个轻量组件被添加到一个容器中，则这些轻量组件的层序是由组件被添加到容器中的顺序来决定的。

如果对此还不太明白，下面的两个小应用程序将会有助于理解。图 2-5 所示的小应用程序有七个按钮，其中四个是重量 AWT 按钮，其他三个是 Swing 轻量按钮。所有的重量按钮都显示在轻量按钮的上面，因为轻量按钮的层序与它们的容器的层序相同。

例 2-6 列出了图 2-5 所示的小应用程序的代码。

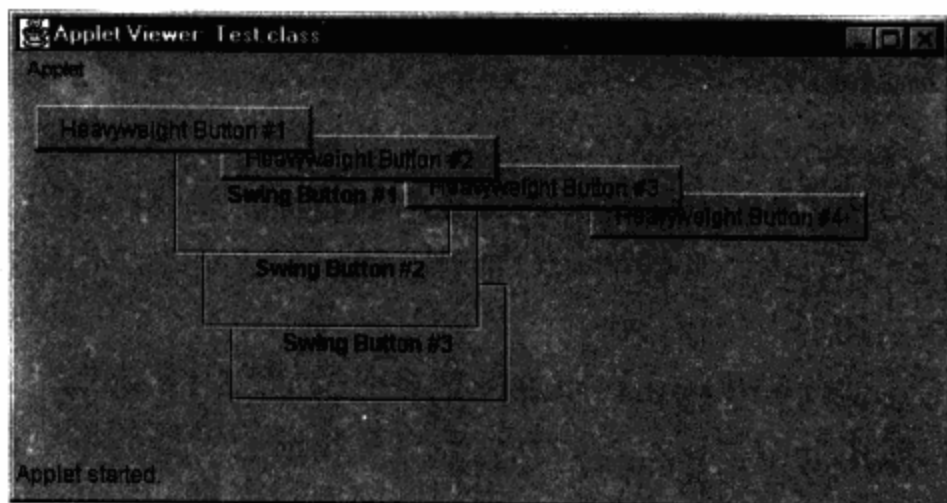


图 2-5 混合使用轻量组件和重量组件

例 2-6 混合使用重量组件和轻量组件

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {
    Button b1, b2, b3, b4;
    JButton jb1, jb2, jb3;

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (null);

        // create heavyweight AWT buttons
        b1 = new Button (" Heavyweight Button # 1");
        b2 = new Button (" Heavyweight Button # 2");
        b3 = new Button (" Heavyweight Button # 3");
        b4 = new Button (" Heavyweight Button # 4");

        // create lightweight Swing buttons
        jb1 = new JButton (" Swing Button # 1");
        jb2 = new JButton (" Swing Button # 2");
        jb3 = new JButton (" Swing Button # 3");

        // set bounds for heavyweight buttons
        b1.setBounds (10, 10, 150, 25);
        b2.setBounds (110, 25, 150, 25);
        b3.setBounds (210, 40, 150, 25);
        b4.setBounds (310, 55, 150, 25);

        // set bounds for lightweight buttons
        jb1.setBounds (85, 25, 150, 65);
        jb2.setBounds (100, 65, 150, 65);
        jb3.setBounds (115, 105, 150, 65);

        // add lightweight buttons
        contentPane.add (jb1);
        contentPane.add (jb2);
        contentPane.add (jb3);

        // add heavyweight buttons
        contentPane.add (b1);
        contentPane.add (b2);
        contentPane.add (b3);
        contentPane.add (b4);
    }
}
```

这个小应用程序把内容窗格的布局管理器设置为 `null`，以便这些按钮可以显式地定位和确定大小，使这些按钮相互重叠。然后，这个小应用程序创建按钮，设置按钮的边界并把每个按钮添加到内容窗格中。

即使轻量按钮在重量按钮之前添加到内容窗格中，轻量按钮也仍在重量按钮下显示。因为轻量组件的层序与它们所在的重量容器的层序相同，所以轻量按钮和它们的容器的层序相同。轻量按钮的容器就是小应用程序的内容窗格。

注意 第一个添加到内容窗格的轻量按钮在其他轻量按钮之上显示。同样，第一个添加到内容窗格的重量按钮在其他重量按钮之上显示。

图 2-6 所示的小应用程序强调了这样一个事实：轻量组件的层序与它们的重量容器的层序相同。这个小应用程序几乎与图 2-5 所示的小应用程序一样，然而，图 2-6 所示的小应用程序把三个轻量按钮放在一个重量面板中。然后再把该面板添加到内容窗格中，使这个重量面板在第二个重量按钮之后，在第三个重量按钮之前。结果，轻量按钮具有与它们所在的面板相同的层序，它们在第二个重量按钮之下，第三个重量按钮之上显示。

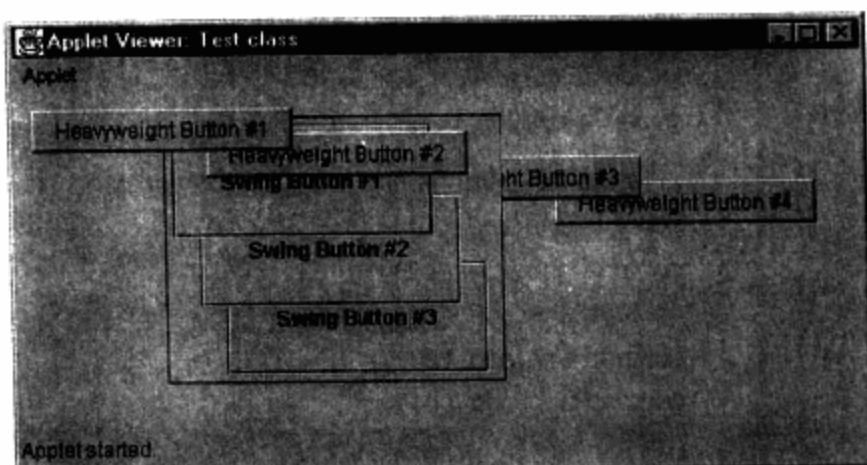


图 2-6 具有与它们的重量容器相同层序的轻量按钮

例 2-7 列出了图 2-6 所示的小应用程序的代码。

例 2-7 控制轻量按钮的层序

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {
    Button b1, b2, b3, b4;
    JButton jb1, jb2, jb3;
    public void init () {
        Container contentPane = getContentPane ();
        Panel p = new BorderedPanel ();

        // set layout managers for content pane and panel
        // to null so their components can be explicitly
        // positioned and sized
        contentPane.setLayout (null);
        p.setLayout (null);

        // create heavyweight AWT buttons
        b1 = new Button (" Heavyweight Button #1");
        b2 = new Button (" Heavyweight Button #2");
        b3 = new Button (" Heavyweight Button #3");
        b4 = new Button (" Heavyweight Button #4");

        // create lightweight Swing buttons
        jb1 = new JButton (" Swing Button #1");
        jb2 = new JButton (" Swing Button #2");
        jb3 = new JButton (" Swing Button #3");

        // set bounds for heavyweights
        b1.setBounds (10, 10, 150, 25);
        b2.setBounds (110, 25, 150, 25);
        b3.setBounds (210, 40, 150, 25);
        b4.setBounds (310, 55, 150, 25);

        // set bounds for lightweights
        jb1.setBounds (5, 5, 150, 65);
        jb2.setBounds (20, 45, 150, 65);
    }
}
```



```

        jb3.setBounds (35, 85, 150, 65);
        // set bounds for panel and add lightweights
        p.setBounds (85, 15, 195, 155);
        p.add (jb1);
        p.add (jb2);
        p.add (jb3);

        // add AWT buttons and panel to content pane
        contentPane.add (b1);
        contentPane.add (b2);
        contentPane.add (p);
        contentPane.add (b3);
        contentPane.add (b4);
    }

    class BorderedPanel extends Panel {
        public void paint (Graphics g) {
            Dimension size = getSize ();

            g.setColor (Color.black);
            g.drawRect (0, 0, size.width-1, size.height-1);
            super.paint (g); // paint lightweights
        }
    }
}

```

例 2-7 的小应用程序实现 `java.awt.Panel` 类的一个扩展 (`BorderedPanel`)，`BorderedPanel` 在面板的外面画了一个黑边框，以使面板可见。

另外还要注意，`BorderedPanel` 类调用 `super.paint()`。无论何时扩展了一个容器并重载了它的 `paint` 方法，都必须显式地调用 `super.paint()`，这样，容器中的轻量组件才能重新绘制^①。如果没有调用 `super.paint()`，则不会重新绘制面板中的轻量 Swing 按钮。

2.3.2 Swing 弹出式菜单

缺省时，Swing 弹出式菜单是轻量组件^②。如果轻量弹出式菜单与重量组件重叠，则弹出式菜单将在该重量组件下面显示。如图 2-7 小应用程序所示。

有些 Swing 组件使用弹出式菜单。Swing 菜单组件就是一种使用弹出式菜单的组件，它在一个菜单被激活时，显示一个弹出式菜单。缺省时，如果一个与某个菜单相关联的弹出式菜单完全处在弹出式菜单所在的窗口中，则弹出式菜单使用轻量组件。图 2-7 所示的小应用程序中与 File 菜单相关联的弹出式菜单是一个轻量组件，所以它在重量组件 AWT 按钮的下面显示。

例 2-8 列出了图 2-7 所示的小应用程序的代码。

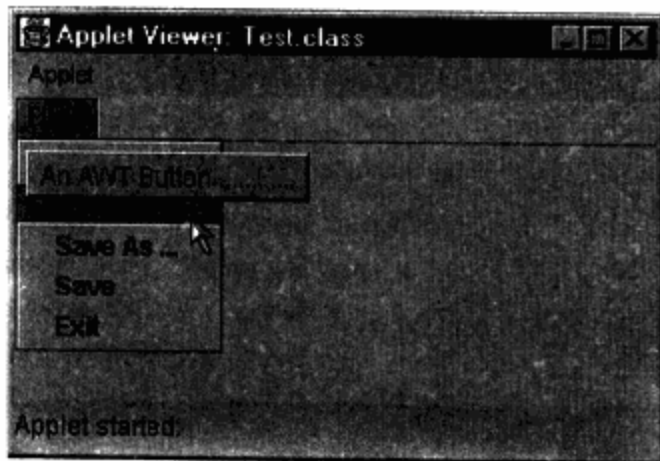


图 2-7 在重量组件下弹出的轻量弹出式菜单

① 有关轻量组件的更多信息，请参见《Graphic Java》第 1 卷。

② 这是一种简化的说法，但适用于此处的讨论。完整的介绍请参见 10.8 节“JPopupMenu”。

例 2-8 在重量组件下面显示的轻量弹出式菜单

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JMenuBar menubar = new JMenuBar ();
        JMenu menu = new JMenu (" File");

        menu.add (" New ...");
        menu.add (" Open ...");
        menu.add (" Save As ...");
        menu.add (" Save");
        menu.add (" Exit");

        contentPane.setLayout (new FlowLayout (FlowLayout.LEFT));
        contentPane.add (new Button (" An AWT Button ....."));

        menubar.add (menu);
        setJMenuBar (menubar);
    }
}

```

这个小应用程序创建了一个菜单条、一个 AWT 按钮和一个菜单。把菜单项添加到菜单中，再把菜单添加到菜单条中，按钮则被添加到小应用程序的内容窗格中。最后，调用 `JApplet.setJMenuBar()`，把菜单条添加到小应用程序中。

幸运的是，Swing 提供了一个机制，它迫使弹出式菜单是重量组件，这样，它们就不会在重量组件下面弹出来。`JPopupMenu` 类提供了一个 `static` 方法，该方法可决定弹出式菜单是重量的还是轻量的^①。

`JPopupMenu.setDefaultLightWeightPopupEnabled()` 以一个 `boolean` 值为参数，这个值指出是把弹出式菜单实例化为轻量的还是把弹出式菜单实例化为重量的，调用 `setDefaultLightWeightPopupEnabled()` 时，如果这个 `boolean` 值为 `true`，则创建的弹出式菜单是轻量的，如果这个 `boolean` 值为 `false`，则创建的弹出式菜单是重量的^②。

图 2-8 所示的小应用程序除了在菜单条被实例化之前调用了 `JPopupMenu.setDefaultLightWeightPopupEnabled(false)` 以外，其余部分都与图 2-7 所示的小应用程序相同。

例 2-9 列出了图 2-8 所示的小应用程序的代码。

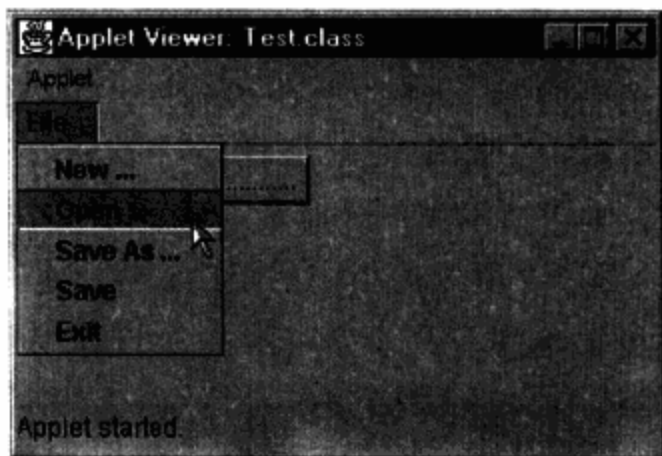


图 2-8 在重量组件之上弹出的重量弹出式菜单

① 某些弹出式菜单即可以指定为轻量的，也可以指定为重量的。

② 这也是简化的说法，但同样适用于这里的讨论。

例 2-9 使用重量弹出式菜单

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        JPopupMenu.setDefaultLightWeightPopupEnabled (false);

        Container contentPane = getContentPane ();
        JMenuBar menubar = new JMenuBar ();
        JMenu menu = new JMenu (" File");

        menu.add (" New ...");
        menu.add (" Open ...");
        menu.add (" Save As ...");
        menu.add (" Save");
        menu.add (" Exit");

        contentPane.setLayout (new FlowLayout (FlowLayout.LEFT));
        contentPane.add (new Button (" An AWT Button ....."));

        menubar.add (menu);
        setJMenuBar (menubar);
    }
}
```

2.3.3 滚动

把重量组件和轻量组件混合使用时所要关心的另一个问题是滚动。

Swing 提供了一个替代 AWT 重量滚动窗格的轻量组件——JScrollPane 组件。由于 JScrollPane 是轻量的，所以，任何添加到 JScrollPane 实例中的重量组件都将在这个滚动窗格之上显示。如果重量组件滚动超出了 JScrollPane 实例的边框，则它就不能正确地显示了。

图 2-9 所示的小应用程序说明了把一个重量组件添加到 JScrollPane 实例中并滚动重量组件使其超出滚动窗格边框的情况。

图 2-9 中上图显示了这个小应用程序刚启动时的样子，图 2-9 中下图显示了滚动窗格滚动后，这个小应用程序的样子。注意，在这两种情况下，AWT 按钮都没有能够正确地显示。

例 2-10 列出了图 2-9 所示的小应用程序的代码。

例 2-10 用 JScrollPane 滚动重量组件

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
```

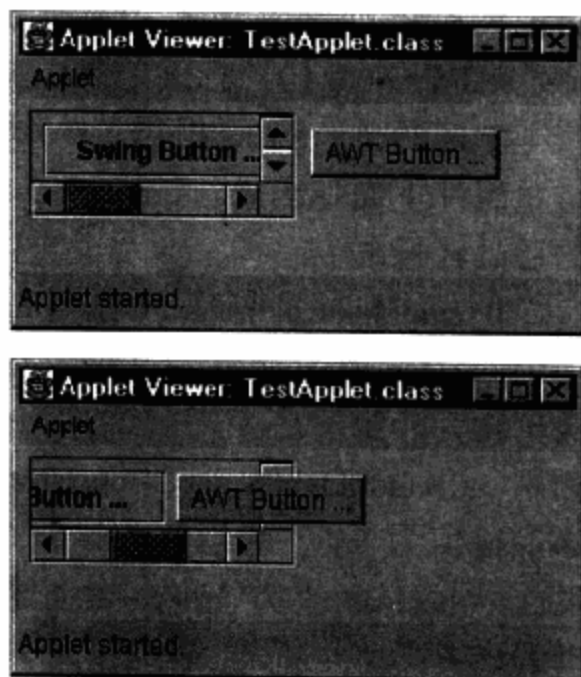


图 2-9 JScrollPane 不能把 AWT 组件放在正确的位置上

```

public Test () {
    JPanel panel = new JPanel ();

    panel.add (new JButton (" Swing Button ..."));
    panel.add (new Button (" AWT Button ..."));

    Container contentPane = getContentPane ();
    JScrollPane scrollPane = new JScrollPane (panel);

    scrollPane.setPreferredSize (new Dimension (125, 50));
    contentPane.setLayout (new FlowLayout (FlowLayout.LEFT));
    contentPane.add (scrollPane);
}

```

图 2-9 所示的小应用程序把一个 Swing 按钮和一个 AWT 按钮添加到一个面板中，这个面板是要滚动的组件。这个小应用程序为滚动窗格设置了首选大小，并把滚动窗格添加到其内容窗格中。

图 2-9 所示的组件效果是我们不想要的。遗憾的是，与弹出式菜单不同，JScrollPane 没有能实例化为重量组件的选项。但是，幸运的是，AWT 的 ScrollPane 组件是一个重量滚动窗格，它和 Swing 的 JScrollPane 几乎完全相同。

图 2-10 示出了与图 2-9 相同的小应用程序，但图 2-10 中的小应用程序用重量 AWT 的 ScrollPane 替代了 Swing 的轻量 JScrollPane。由于 AWT 滚动窗格是重量的，所以它们滚动轻量组件和重量组件都没有问题。

例 2-11 列出了图 2-10 所示的小应用程序的代码

例 2-11 使用 AWT 的 ScrollPane 来滚动重量组件

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public Test () {
        JPanel panel = new JPanel ();

        panel.add (new JButton (" Swing Button ..."));
        panel.add (new Button (" AWT Button ..."));

        Container contentPane = getContentPane ();
        SizedScrollPane scrollPane = new SizedScrollPane ();

        scrollPane.add (panel);

        contentPane.setLayout (new FlowLayout (FlowLayout.LEFT));
        contentPane.add (scrollPane);
    }
}

class SizedScrollPane extends ScrollPane {
    public Dimension getPreferredSize () {
        return new Dimension (125, 50);
    }
}

```

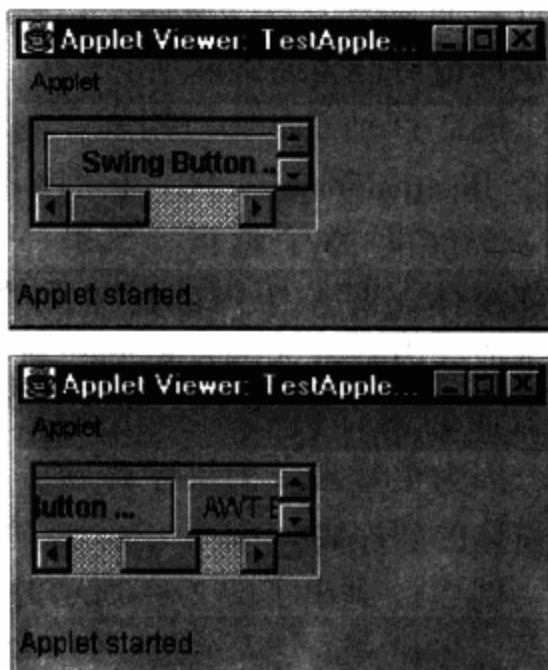


图 2-10 java.awt.ScrollPane 滚动 AWT 组件和 Swing 组件

注意 在例 2-11 列出的小应用程序中实现了 `java.awt.ScrollPane` 的一个扩展，以便把滚动窗格的大小设置为首选尺寸。有关 Swing 组件与 AWT 组件在设置首选尺寸方面的差别的更多信息，请参见 4.2.2 节“最小尺寸、最大尺寸和首选尺寸”。

2.3.4 内部窗体

Swing 的内部窗体是包含在桌面窗格中的窗体（参见第 15 章“内部窗体和桌面窗格”），Swing 的内部窗体是轻量组件，如果把重量组件添加到一个内部窗体中，则这个窗体很可能会遇到麻烦。

图 2-11 所示的小应用程序包含两个 `JInternalFrame` 实例。它们都包含一个重量 AWT 画布。如果一个内部窗体与另一个内部窗体重叠，则下面的内部窗体的重量画布将会使上面的内部窗体的一部分变模糊，因为重量画布的层序比轻量内部窗体的层序高。

例 2-12 列出了图 2-11 所示的小应用程序的代码。

例 2-12 把重量组件添加到 Swing 内部窗体中

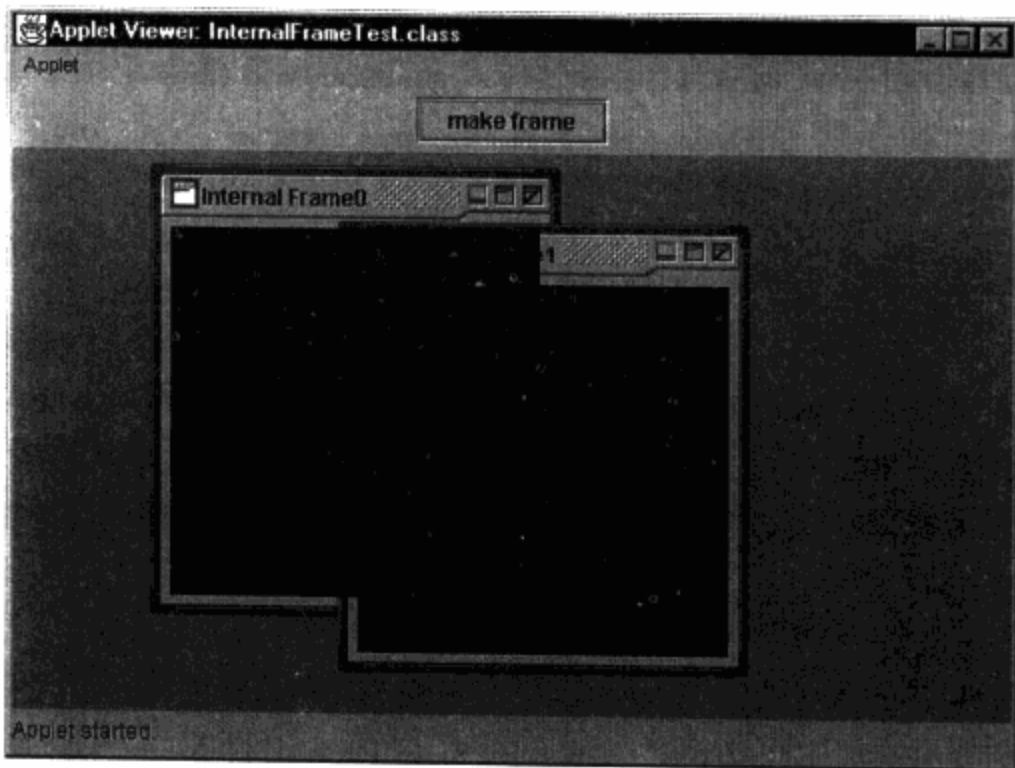


图 2-11 具有重量组件的内部窗体

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class InternalFrameTest extends JApplet {
    JDesktopPane dtp = new JDesktopPane();

    public void init() {
        JPanel controlPanel = new ControlPanel(dtp);
        Container contentPane = getContentPane();
        JPanel centerPanel = new JPanel();

        contentPane.setLayout(new BorderLayout());
        contentPane.add(controlPanel, BorderLayout.NORTH);
        contentPane.add(dtp, BorderLayout.CENTER);
    }

    class ControlPanel extends JPanel {
        private static int cnt = 0;

        public ControlPanel(final JDesktopPane dtp) {
            JButton b = new JButton("make frame");
            add(b);

            b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    JInternalFrame jif = new JInternalFrame();
                    Container contentPane = jif.getContentPane();
                }
            });
        }
    }
}
```

```

        jif.setLocation (10, 50);
        jif.setTitle (" Internal Frame" + cnt + + );
        jif.setResizable (true);
        jif.setMaximizable (true);
        jif.setClosable (true);
        jif.setVisible (true);
        jif.setIconifiable (true);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (new ColoredCanvas (), " Center");
        jif.pack ();

        dtp.add (jif, 2); // add at layer 2
    }

    });
}

class ColoredCanvas extends Canvas {
    public void paint (Graphics g) {
        Dimension sz = getSize ();
        g.setColor (Color.blue);
        g.fillRect (0, 0, sz.width, sz.height);

        public Dimension getPreferredSize () {
            return new Dimension (200, 200);
        }
    }
}

```

Swing 提示

混合使用 AWT 组件和 Swing 组件的原则

一般不提倡把 Swing 轻量组件与 AWT 重量组件混合使用。大多数情况下, 这不会是一个问题, 因为 Swing 对所有 AWT 组件都提供了替代的轻量组件。对已有的、使用 AWT 组件的小应用程序或应用程序, 最好的方法是用 Swing 的相应组件来替代 AWT 组件。如果不能替代, 则必须遵守如下原则:

- 1) 如果轻量组件必须在重量组件之上显示, 则不要在一个容器中混合使用轻量组件和重量组件。
- 2) 如果弹出式菜单与重量组件重叠, 则必须强迫弹出式菜单成为重量组件。
- 3) 不要把重量组件添加到一个 JScrollPane 实例中, 而应该把重量组件添加到一个 java.awt.ScrollPane 实例中。
- 4) 不要把重量组件添加到 Swing 内部窗体中。

2.4 Swing 和线程

大多数情况下, Swing 是线程不安全的, 即只能从单线程来访问 Swing 组件。首先, 我们要讨论为什么 Swing 是线程不安全的, 然后介绍在 Swing 开发过程中单线程设计所带来的结果。

让我们面对这个事实, 甚至在 Java 中, 开发多线程的应用程序也是不容易的。设计一个线程安全的工具包就更不是一个简单的事情。例如, 确定如何同步对类的访问就是一个复杂的

任务^①。同样，扩展线程安全的类需要较高的技术，对非线程编程高手的开发人员（大多数开发人员都属此范围）是充满危险的。Swing 是线程不安全的一个主要原因是为了简化扩展组件的任务。

Swing 是线程不安全的另一个原因是由于获取和释放锁定及恢复状态所带来的开销。使用线程安全 GUI 工具包的所有应用程序（无论它们是否是多线程的）都必须付出同样的性能代价。

线程的使用增加了调试、测试、维护和扩展的困难度。例如，测试和维护等通常已经很艰苦的工作对于大多数多线程应用程序就更困难了，有时甚至是不可能的。

有些 Swing 组件方法确实支持多线程访问。例如，JComponent 的 `repaint`、`revalidate` 和 `invalidate` 等方法都对放在事件派发线程上的请求进行排队，因此，可从任何线程中调用这些方法。另外，可以从多个线程把监听器添加到事件监听器列表（参见 6.2 节“事件监听器列表”）中或从列表中删掉。最后，有些组件方法是同步的。例如，`JCheckBoxMenuItem.setState()` 是同步的，因此，可以从多线程中调用它。

2.4.1 Swing 单线程设计的结果

Swing 单线程设计的主要结果是：大多数情况下，只能从事件派发线程中访问将要在屏幕上绘制的 Swing 组件。

事件派发线程是调用 `paint` 和 `update` 等回调方法的线程，而且，它还是事件监听器接口中定义的事件处理方法。例如，`ActionListener` 和 `PropertyListener` 接口的实现使它们的 `actionPerformed` 方法和 `propertyChange` 方法在事件派发线程中调用。

技术上说，在 Swing 组件的对等组件创建之前（指可在屏幕上绘制之前）^②，它们可以从多个线程中访问。例如，可以在一个小应用程序的 `init` 方法中构造和操纵组件，只要在操纵它们之前，还没有使它们成为可见的。

2.4.2 SwingUtilities 类的 `invokeLater` 和 `invokeAndWait` 方法

由于 AWT 和 Swing 都是事件驱动工具包，所以在回调方法中更新可见的 GUI 就是很自然的事。例如，如果在一个按钮激活，项目列表需要更新时，则通常在与该按钮相关联的事件监听器的 `actionPerformed` 方法中来实现该列表的更新。

然而，有时可能需要从事件派发线程以外的线程中更新 Swing 组件。例如，如果上述项目列表中包含了很多来自数据库或 Internet 的数据，则可能在按钮激活后还要等一段时间才能看到更新的列表。如果信息的获取是在 `actionPerformed` 中实现的，则按钮仍保持按下的状态，直到对 `actionPerformed` 的调用返回，不仅按钮的弹起需要一段时间，而且一般来说，耗时较长的操作也不应当在事件处理方法中执行，因为在事件处理方法返回之前，其他的事件不能派发。

有时，在独立的线程上执行耗时的操作可能更好，这将允许立即更新用户界面和释放事件派发线程去派发其他的事件。幸运的是，Swing 提供了两种机制，它们都支持这种想法。

SwingUtilities 类提供了两个方法：`invokeLater` 和 `invokeAndWait`，它们都使事件派发线程上的可运行对象排队。当可运行对象排在事件派发队列的队首时，就调用其 `run` 方法。其效果是允许事件派发线程调用另一个线程中的任意一个代码块。

① 参见 Lea, Doug, “java 中的并发编程”, Addison-Wesley, 1997。

② 对等组件是用 `addNotify` 方法创建的。

1. SwingUtilities.invokeLater

在介绍 `invokeLater` 和 `invokeAndWait` 方法之前，我们首先来看一个小应用程序，由于是从事件派发线程以外的线程中更新 Swing 组件，所以该小应用程序运行不正常。图 2-12 所示的小应用程序有一个按钮和一个进度条。当激活按钮后，就开始模仿获取信息的长操作。当获取了信息（即一个 `integer` 值）后，就用该信息来更新小应用程序的进度条。

图 2-12 左图显示的是这个小应用程序的初始状态。图 2-12 右图显示的则是当激活 `start` 按钮后，这个小应用程序的样子，此时，已获取了信息，也更新了进度条。

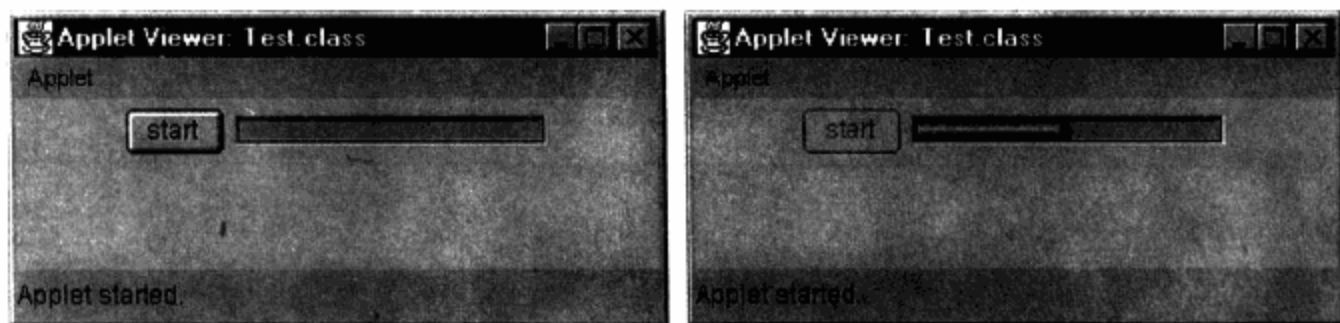


图 2-12 从多个线程中更新 Swing 组件（用 Mac 界面样式显示）

小应用程序把一个动作监听器添加到该按钮中，该监听器创建一个新线程，这个线程不断收到信息并更新进度条。每隔半秒会获取一次信息，而且这个线程会获得一个对这个小应用程序进度条的引用。

```
public class Test extends JApplet {
    ...
    public void init () {
        ...
        startButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                GetInfoThread t = new GetInfoThread (Test.this);
                t.start ();

                // this is ok, because actionPerformed
                // is called on the event dispatch thread
                startButton.setEnabled (false);
            }
        });
    }
    ...
    class GetInfoThread extends Thread {
        Test applet;

        public GetInfoThread (Test applet) {
            this.applet = applet;
        }

        public void run () {
            while (true) {
                try {
                    // simulate "lengthy" information retrieval
                    Thread.currentThread ().sleep (500);

                    // this is not ok, because it is not called
                    // on the event dispatch thread
                    applet.getProgressBar ().setValue (Math.random () * 100);
                } catch (InterruptedException e) {}
            }
        }
    }
}
```



```

    |
    | catch (InterruptedException e) |
    |     e.printStackTrace ();
    |
    |
    |
    |

```

在该按钮的监听器启动上述线程后，监听器把按钮的允许状态设置为 `false`。由于在事件派发线程上调用 `actionPerformed` 方法，所以，这是一个有效的操作。但是，在 `GetInfoThread` 中设置进度条是一个危险的做法，因为事件派发线程以外的线程将更新进度条。

例 2-13 列出了图 2-12 所示的小应用程序的完整的代码。

例 2-13 从另一个线程更新组件的错误方法

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    JProgressBar pb = new JProgressBar ();

    public void init () {
        Container contentPane = getContentPane ();
        final JButton startButton = new JButton (" start");

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (startButton);
        contentPane.add (pb);

        startButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                GetInfoThread t = new GetInfoThread (Test.this);
                t.start ();

                // this is ok, because actionPerformed
                // is called on the event dispatch thread
                startButton.setEnabled (false);
            }
        });
    }

    public JProgressBar getProgressBar () {
        return pb;
    }
}

class GetInfoThread extends Thread {
    Test applet;

    public GetInfoThread (Test applet) {
        this.applet = applet;
    }

    public void run () {
        while (true) {
            try {
                // simulate " lengthy" information retrieval
                Thread.currentThread ().sleep (500);

                // this is not ok, because it is not called

```

```

        // on the event dispatch thread
        applet.getProgressBar ().setValue (Math.random () * 100);
    }
    catch (InterruptedException e) {
        e.printStackTrace ();
    }
}

```

更新例 2-13 所示的小应用程序中的进度条的正确方法是使用 `SwingUtilities.invokeLater` (或 `invokeAndWait`)。下面列出的 `GetInfoThread` 类的构造方法被修改了以便实例化一个可运行的对象, 该对象获取对小应用程序进度条的引用并更新进度条的值。`GetInfoThread` 类的 `run` 方法调用 `SwingUtilities.invokeLater` 并把对进度条的引用传送给可运行对象。

```

class GetInfoThread extends Thread {
    Runnable runnable;
    int value;

    public GetInfoThread (final Test applet) {
        runnable = new Runnable () {
            public void run () {
                JProgressBar pb = applet.getProgressBar ();
                pb.setValue (value);
            }
        };
    }

    public void run () {
        while (true) {
            try {
                Thread.currentThread ().sleep (500);

                // This is okay because the runnable's run ()
                // will be invoked on the event dispatch thread
                value = (int) (Math.random () * 100);
                SwingUtilities.invokeLater (runnable);
            }
            catch (InterruptedException e) {
                e.printStackTrace ();
            }
        }
    }
}

```

例 2-14 是例 2-13 所列的小应用程序的修改版。

例 2-14 从另一个线程中更新组件的正确方法

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private JProgressBar pb = new JProgressBar ();

    public void init () {
        Container contentPane = getContentPane ();
    }
}

```

```

        final JButton startButton = new JButton (" start");
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (startButton);
        contentPane.add (pb);

        startButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                GetInfoThread t = new GetInfoThread (Test.this);
                t.start ();

                // this is okay, because actionPerformed
                // is called on the event dispatch thread
                startButton.setEnabled (false);
            }
        });
    }

    public JProgressBar getProgressBar () {
        return pb;
    }
}

class GetInfoThread extends Thread {
    Runnable runnable;
    int value;

    public GetInfoThread (final Test applet) {
        runnable = new Runnable () {
            public void run () {
                JProgressBar pb = applet.getProgressBar ();
                pb.setValue (value);
            }
        };
    }

    public void run () {
        while (true) {
            try {

                Thread.currentThread ().sleep (500);

                // This is okay because the runnable's run ()
                // is invoked on the event dispatch thread
                value = (int) (Math.random () * 100);
                SwingUtilities.invokeLater (runnable);

            } catch (InterruptedException e) {
                e.printStackTrace ();
            }
        }
    }
}

```

2. SwingUtilities. InvokeAndWait

与 `invokeLater` 一样, `SwingUtilities.InvokeAndWait` 也把可运行对象排入事件派发线程的队列中。虽然, `invokeLater` 在把可运行对象放入队列后就返回, 而 `InvokeAndWait` 一直等待直到已启动了可运行对象的 `run` 方法才返回。如果在另一个操作能够在另一个线程上执行之前必须从一个组件获取信息, 则 `InvokeAndWait` 方法是很有用的。

例如，例 2-14 列出的小应用程序总是更新进度条的值而不管该新值是否与当前的值相同。如果只在新值与当前值不同时才更新进度条的值，则效率更高。修改这个小应用程序，使得这个小应用程序只在新值与当前值不同时才更新进度条的值，这将使我们有机会进一步介绍 `InvokeAndWait` 方法。

首先，修改 `GetInfoThread` 类以创建两个可运行的对象：一个对象获得进度条当前的值，另一个对象用于设置进度条的值。

```
class GetInfoThread extends Thread {
    Runnable getValue, setValue;
    int value, currentValue;

    public GetInfoThread (final Test applet) {
        getValue = new Runnable () {
            public void run () {
                JProgressBar pb = applet.getProgressBar ();
                currentValue = pb.getValue ();
            }
        };
        setValue = new Runnable () {
            public void run () {
                JProgressBar pb = applet.getProgressBar ();
                pb.setValue (value);
            }
        };
    }
}
```

接着，使用 `invokeAndWait()` 来修改 `GetInfoThread` 类的 `run` 方法以获取进度条的当前值。

```
public void run () {
    while (true) {
        try {
            // simulate "lengthy" information retrieval
            Thread.currentThread ().sleep (500);

            value = (int) (Math.random () * 100);

            try {
                SwingUtilities.invokeAndWait (getValue);
            }
            catch (InvocationTargetException ite) {
                ite.printStackTrace ();
            }
            catch (InterruptedException ie) {
                ie.printStackTrace ();
            }

            if (currentValue != value) {
                SwingUtilities.invokeLater (setValue);
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace ();
        }
    }
}
```

`SwingUtilities.invokeAndWait()` 获取进度条的当前值，`invokeLater()` 则设置进度条的值。对

InvokeAndWait 的调用直到 getValue 可运行对象的 run 方法返回后才返回。

SwingUtilities.invokeLater 可能会弹出下面两个异常信息之一：InterruptedException 或 InvocationTargetException。无论何时使用 invokeLater() 都必须捕捉这些异常，否则，调用 invokeLater() 的方法中必须有一个 throw 子句。

例 2-15 显了这种方法的完整代码。

例 2-15 使用 SwingUtilities.invokeLater()

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*; // for InvocationTargetException

public class Test extends JApplet {
    private JProgressBar pb = new JProgressBar();

    public void init() {
        Container contentPane = getContentPane();
        final JButton startButton = new JButton("start");

        contentPane.setLayout(new FlowLayout());
        contentPane.add(startButton);
        contentPane.add(pb);

        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                GetInfoThread t = new GetInfoThread(Test.this);
                t.start();

                // this is okay because actionPerformed
                // is called on the event dispatch thread
                startButton.setEnabled(false);
            }
        });
    }

    public JProgressBar getProgressBar() {
        return pb;
    }

    class GetInfoThread extends Thread {
        Runnable getValue, setValue;
        int value, currentValue;

        public GetInfoThread(final Test applet) {
            getValue = new Runnable() {
                public void run() {
                    JProgressBar pb = applet.getProgressBar();
                    currentValue = pb.getValue();
                }
            };
        }

        setValue = new Runnable() {
            public void run() {
                JProgressBar pb = applet.getProgressBar();
                pb.setValue(value);
            }
        };
    }
}
```

```

|
public void run () {
    while (true) {
        try {
            Thread.currentThread().sleep (500);

            // This is okay because the getValue' s run ()
            // is invoked on the event dispatch thread
            value = (int) (Math.random () * 100);

            try {
                SwingUtilities.invokeLater (getValue);
            }
            catch (InvocationTargetException ite) {
                ite.printStackTrace ();
            }
            catch (InterruptedException ie) {
                ie.printStackTrace ();
            }

            if (currentValue != value) {
                SwingUtilities.invokeLater (setValue);
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace ();
        }
    }
}
|

```

invokeLater () 和 invokeAndWait () 之间一个重要的区别是：可以从事件派发线程中调用 invokeLater ()，却不能从事件派发线程中调用 invokeAndWait。从事件派发线程调用 invokeAndWait () 所带来的问题是：invokeAndWait () 锁定调用它的线程，直到可运行对象从事件派发线程中派发出去并且该可运行对象的 run 方法激活。如果从事件派发线程调用 invokeAndWait ()，则将发生线程死锁的情况，因为 invokeAndWait () 正在等待事件派发，但是，由于是从事件派发线程中调用 invokeAndWait () 的，所以，直到 invokeAndWait () 返回后事件才能派发。

Swing 提示

使用 SwingUtilities.invokeLater () 和 SwingUtilities.invokeLaterAndWait () 从事件派发线程之外的线程访问组件

由于 Swing 是线程不安全的，所以，从事件派发线程之外的线程访问 Swing 组件是不安全的。SwingUtilities 类提供了两个用于执行事件派发线程中代码的方法，这两种方法是 invokeLater 和 invokeAndWait。

注意 可以从事件派发线程调用 SwingUtilities.invokeLater，却不能从事件派发线程调用 SwingUtilities.invokeLaterAndWait。如果从事件派发线程调用 SwingUtilities.invokeLaterAndWait，则将发生线程死锁。因为 invokeAndWait 正在等待可运行对象被从事件派发线程中派发出去，但是在调用 SwingUtilities.invokeLaterAndWait 的线程返回前事件不能派发。

2.5 本章回顾

Swing 的设计目标之一是为实现小应用程序和应用程序的完整性制定一些约定。大多数情况下，这个目标已经达到了。Swing 小应用程序和应用程序含有 `JRootPane` 的一个实例，这意味着不能把组件直接添加到 `JApplet` 或 `JFrame` 的实例中，也不能显式地为 `JApplet` 或 `JFrame` 的实例设置布局管理器。组件应该添加到根窗格的内容窗格中，同理，必须为内容窗格设置布局管理器而不是为小应用程序和应用程序设置布局管理器。幸运的是，无论何时组件直接添加、或显式地为小应用程序或窗体设置了布局管理器，`JApplet` 和 `JFrame` 都会弹出带错误的异常消息。

把 Swing 实现为线程不安全的决定是肯定会遭到反对的。毕竟，Java 语言本身就内置了多线程特性，因此，就会有人主张应当以线程安全的模式实现 Swing。

然而，正是因为 Java 内置了对多线程的支持，但这并不意味着在 Java 中实现安全的多线程小应用程序或应用程序是一件简单的事情，更不用提工具包了。事实正相反，以线程安全的方式实现复杂的小应用程序和应用程序是相当困难的。另外，大多数开发人员不精通开发复杂的多线程代码。当多线程被引入到面向对象语言中以后，人们遇到的较困难的领域之一就是如何扩展线程安全的类。相比之下，Swing 开发人员使用的单线程方法使得类很容易扩展。

总之，禁止从事件派发线程外的其他线程访问 Swing 组件的决定是正确的，它产生了一个较容易扩展的、较简单的工具包。另外，除事件派发线程外的其他线程可以调度在事件派发线程上实现的可运行对象。

Swing 是一个可靠的、工业标准的用户界面工具包，比 AWT 大有改进。但是，与任何重要的软件一样，Swing 很容易学习，但也有程序错误。

第3章 Swing 组件的体系结构

轻量 Swing 组件把它们的界面样式 (look and feel) 交给一个 UI 代表来处理, 这个 UI 代表负责绘制组件 (即 look) 并处理组件的事件 (即 feel)。可在构造组件之时或之后, 把 UI 代表插入这个组件中。“插入式界面样式”这个术语在 1.4 节中介绍过。

Swing 的插入式界面样式由一个基于 Smalltalk 的“模型-视图-控制器 (Model-View-Controller, MVC)”设计的组件体系结构和用于管理界面样式的下层构件组成。前者是本章重点, 首先我们给出典型的 MVC 的概览, 然后再介绍 Swing MVC 的实现。后者稍后将在第 7 章中介绍。

3.1 典型的“模型-视图-控制器”体系结构

MVC 体系结构是为那些需要为同样的数据提供多个视图的应用程序而设计的。MVC 把应用程序分为三种对象类型:

- 模型: 维护数据并提供数据访问方法。
- 视图: 绘制模型的部分数据或所有数据的可视图。
- 控制器: 处理事件。

模型负责维护数据, 例如, 一个笔记本应用程序将把文档文本存储在模型中。模型通常提供访问和修改数据的方法。当模型变化时, 这个模型还把事件发送给已登记的视图, 对此, 视图根据模型的变化来更新自己。

视图负责提供模型的部分数据的可视图。例如, 一个笔记本应用程序通过显示存储在模型中的部分文本或所有文本来提供当前文档的一个视图。

控制器为视图处理事件。鼠标和动作监听器等 AWT 和 Swing 监听器都是 MVC 控制器。前面提到的笔记本应用程序应该有鼠标和键盘监听器, 以便适时地改变模型或视图。

MVC 需要很强的设计功能。首先, 应当可以把多个视图和控制器插入到单个模型中, 这是 Swing 插入式界面样式的基础。

其次, 当模型改变时, 模型的视图能够自动地得到通知; 在一个视图中改变模型的属性, 将导致模型其他的视图也随之更新。

最后, 由于模型独立于视图, 所以, 不需要修改模型来适应新类型的视图或控制器。

3.1.1 插入式视图和控制器

Swing (和 AWT) 容器把定位它们所包含的组件及确定这些组件的大小的工作委托给一个布局管理器。布局管理器封装了布局组件的策略。例如, FlowLayout 布局管理器的策略是用组件的首选大小来安排组件的大小, 并以从左到右、从上到下的顺序定位组件。

封装的策略使它们是可插入的; 例如, 布局管理器可以在编译时, 也可以在运行时刻插入到容器中。

通过封装在视图中可视地表示数据的策略及封装控制器中处理事件的策略, MVC 体系结构提供了可插入视图和控制器。就像布局管理器可以插入到 AWT 和 Swing 组件中一样, 视图和控制器也可以插入到模型中。

3.1.2 视图更新

Swing（和 AWT）事件由向事件源登记了的事件监听器来处理。例如，按钮的激活事件由一个对象所处理，这个对象实现 ActionListener 接口，并且通过调用这个按钮的 addActionListener 方法向这个按钮进行了登记^①

事件源和监听器是 Observer 样式的一个例子，Observer 样式允许单个对象在所观察的对象修改时通知许多观察器。Observer 样式需要在被观察对象与它的观察器之间有一个很小的接口区。例如，上面描述的动作监听器可以是任何类型的对象，只要它实现 ActionListener 接口。而且，按钮除了知道如何和何时通知观察器之外，对观察器一无所知。

当模型改变时，MVC 体系结构使用 Observer 样式来通知视图。模型可以有許多视图，所有的视图通常都用模型的通知来同步。而且，任何类型的视图都可以在不使模型本身有任何变化的情况下观察一个模型。

图 3-1 示出了一个典型的 MVC 实现的信息流，并说明了在模型变化时，视图如何更新。

事件由控制器处理，控制器根据事件的类型来改变模型或一个或多个视图。

模型维护一个视图列表，这些视图为获得模型变化通知已经向模型登记过了。当模型发生变化时，该模型通知已向此模型登记的每个视图。视图通常从该模型中获得信息以进一步澄清这个事件，接着再更新它们自己。

Swing 提示

MVC 的优点

很久以来，MVC 体系结构一直是建立 Smalltalk 应用程序的基础。

面向对象开发的最基础的方面是确认抽象并在类中封装抽象。例如，一个工资册应用程序可能确认雇员、工资等抽象。在类中封装抽象允许在对象间建立松散的联系，这样就减少了依赖性，增加了灵活性和再使用性。

MVC 封装了三个在大多数图形应用程序都存在的通用抽象：模型、视图和控制器。通过封装其他体系结构的优秀特性，MVC 应用程序比相应的传统应用程序更灵活和更具再使用性。

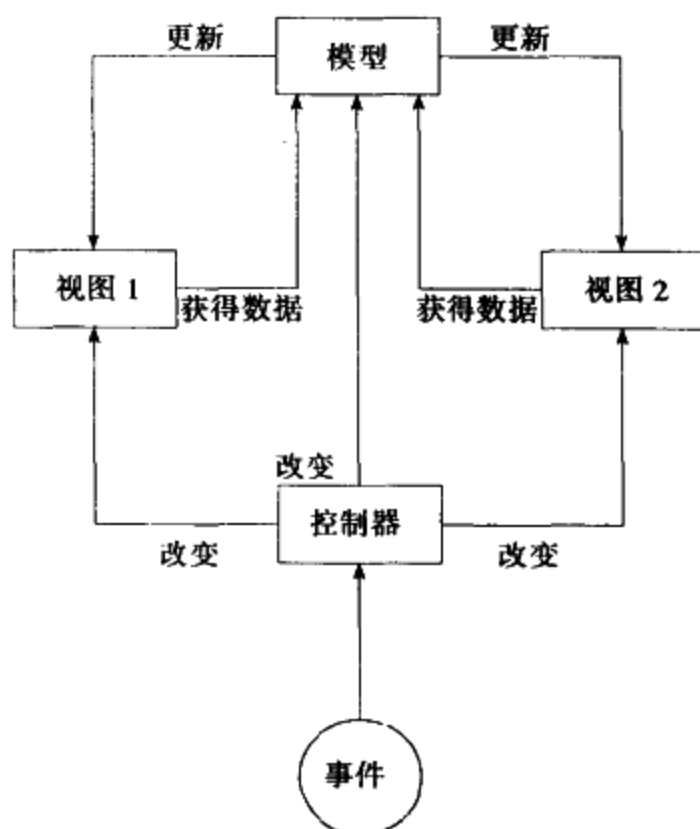


图 3-1 典型的 MVC 通信

3.2 Swing MVC

Swing MVC 是典型 MVC 的专业版本，其设计目的是支持插入式界面样式而不是通用应用程序。Swing 轻量组件由下面的对象组成：

- 一个维护组件的数据模型。

^① 处理按钮动作事件的例子，请参见 8.4 节“JButton 事件”

- UI 代表，它是一个带事件处理监听器的视图。
- 一个扩展 JComponent 的组件[○]。

Swing 模型可以直接对应典型的 MVC 模型；这两个模型都维护数据并提供数据访问方法，在它们发生变化时，它们都通知监听器。

Swing 组件把它们的界面样式交给一个 UI 代表来处理。UI 代表与典型的 MVC 中的视图/控制器组合相对应。从现在开始，控制器又称作监听器。

Swing 监听器通常作为 UI 代表的内部类来实现。例如，一个滑杆的 UI 代表实现一个响应模型变化的变化监听器。这个变化监听器是作为 BasicSliderUI 的内部类实现的：

```
// From javax.swing.plaf.basic.BasicSliderUI.java:
public class BasicSliderUI extends SliderUI {
    ...
    // installUI is called when a UI is being installed
    // for a component
    public void installUI (JComponent c) {
        ...
        changeListener = createChangeListener (slider);
        ...
        installListeners (slider);
    }
    ...
    protected ChangeListener
        createChangeListener (JSlider slider) {
        return new ChangeHandler ();
    }
    ...
    protected void installListeners (JSlider slider) {
        ...
        slider.getModel ().addChangeListener (ChangeListener);
        ...
    }
    ...
    public class ChangeHandler implements ChangeListener {
        public void stateChanged (ChangeEvent e) {
            if (! isDragging) {
                calculateThumbLocation ();
                slider.repaint ();
            }
        }
    }
    ...
}
```

BasicSliderUI 创建了 ChangeHandler 的一个实例，该实例计算滑杆的滑块（即滑柄）的位置并重画该滑杆。

根据组件所表现的复杂程度，组件代表可以有許多处理事件的内部类监听器。例如，BasicSliderUI 类实现六个内部类监听器，如图 3-2 所示。

○ 参见第 4 章“JComponent 类”

3.2.1 Swing 组件

组件为开发人员提供了一个 API 以操纵构成一个 Swing 组件的对象集。组件间接地创建它们的 UI 代表，并在适当的时候把任务交给这些 UI 代表。参见 3.2.6 节“组件 UI 的案例”，参见“UI 代表绘制”中有关创建 UI 代表以及“安装一个 UI 代表”组件把绘制任务交给它们的 UI 代表的有关介绍。

通过提供传递方法和通过传送模型事件，组件还使它们的模型对开发人员透明。

1. 模型传递方法

Swing 组件为它们的模型提供传递方法，以便开发人员不需要直接访问模型来修改或查询状态。例如，下面列出的 JSlider 类的方法显示了滑杆是如何传递它们模型的最小值的。

```
// From JSlider.java, pass-through model methods:
public int getMinimum () {
    return getModel ().getMinimum ();
}
public void setMinimum (int minimum) {
    int oldMin = getModel ().getMinimum ();
    getModel ().setMinimum (minimum);
    firePropertyChange (" minimum",
        new Integer (oldMin), new Integer (minimum));
}
```

JSlider. setMinimum () 在设置最小值后激发一个属性变化事件。组件模型的所有属性（如一个滑杆的最小值和最大值）都应该激发属性变化事件。

2. 传送模型事件

Swing 组件还把模型事件传送给一个已向组件登记过的监听器。例如，一个滑杆作为一个变化监听器向其模型登记。当这个滑杆的模型激发了一个变化事件时，这个滑杆接着把一个变化事件发送给自己的变化监听器。JSlider 类实现一个变化监听器，它只把一个状态变化事件发送给滑杆的变化监听器。与组件 UI 一样，JSlider 等组件类常常在内部类中封装事件处理。

下面列出了大大简化了的 JSlider 类进行监听的代码，其中说明了滑杆把状态变化发送给它们的监听器以响应模型状态的变化的方法（与其他 Swing 组件类的处理方法类似）。

```
// From JSlider.java:
public class JSlider extends JComponent
    implements SwingConstants, Accessible {
    ...
    protected ChangeListener changeListener =
        createChangeListener ();
    ...
    public JSlider (int orientation, int min,
        int max , int value) {
        ...
        sliderModel.addChangeListener (changeListener);
        ...
    }
    ...
    public void addChangeListener (ChangeListener l) {
        listenerList.add (ChangeListener, class, l);
    }
    public void remove Change Listener (change Listener l) {
```

```

listenerList.remove (Change Listener.class, 1);
|
...
protected ChangeListener createChangeListener () {
    return new ModelListener ();
}
|
...
private class ModelListener
    implements ChangeListener, Serializable {
    public void stateChanged (ChangeEvent e) {
        // fire event to change listener registered
        // with addChangeListener () listed above
        fireStateChanged ();
    }
}

```

JSlider 构造方法把一个 JSlider.ModelListener 实例添加到滑杆的模型中。JSlider.ModelListener 通过调用 JSlider.fireStateChanged () 方法来对模型变化做出反应。JSlider.fireStateChanged () 方法把一个变化事件发送给滑杆的监听器。

3.2.2 静态认识

轻量 Swing 组件的实现方式尽量与组成它们的 MVC 结构的对象的实现方式相类似。例如，Swing 按钮由 JButton 类、ButtonUI 类及其他对象组成。其他轻量 Swing 组件与此相同，以相似的

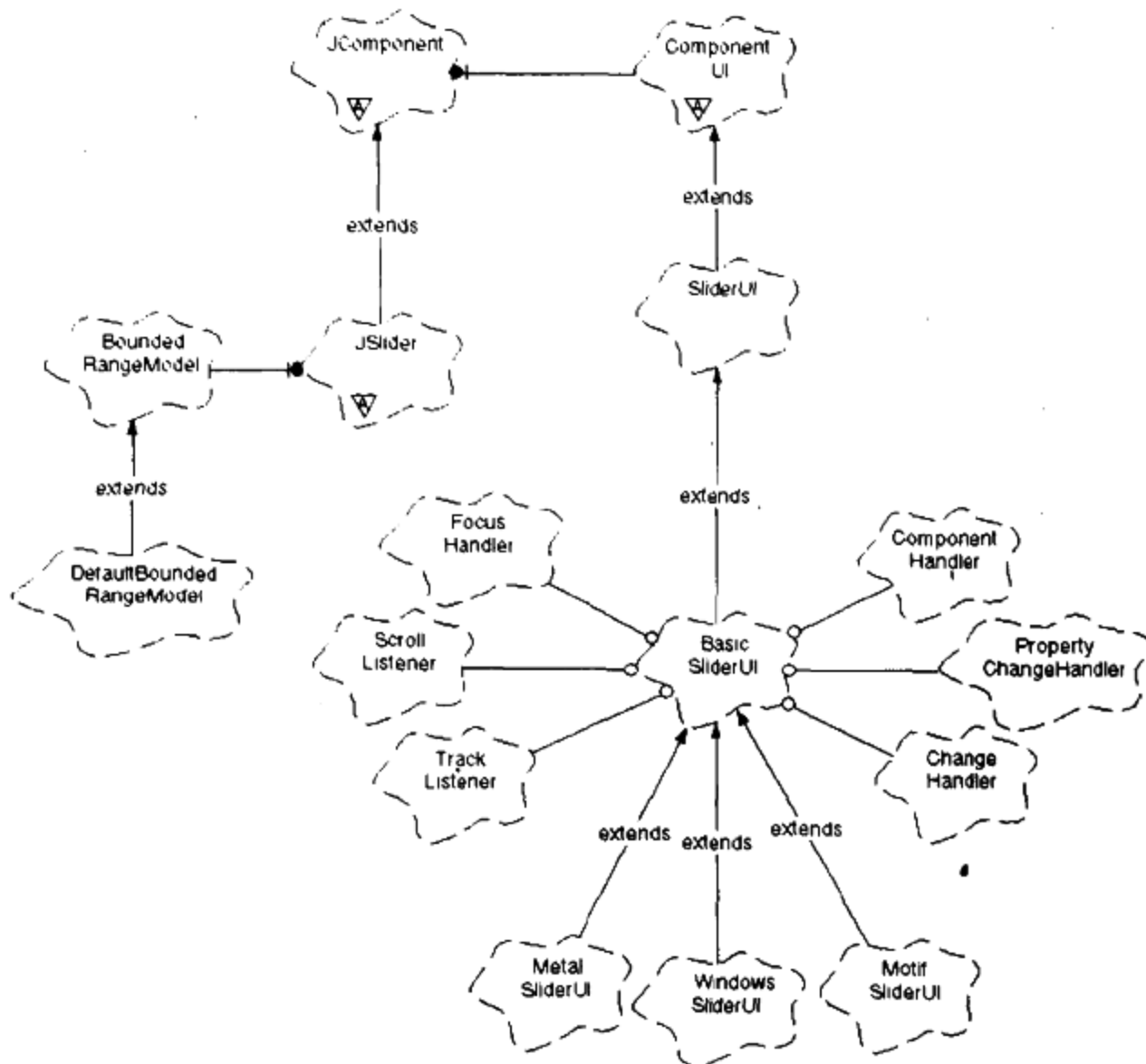


图 3-2 滑杆组件类

类名实现相似的功能，如：JLabel 与 LabelUI、JCheckBox 与 CheckBoxUI、JTree 与 TreeUI 等等。

图 3-2 示出了组成 Swing 滑杆的类的类图。由于轻量 Swing 组件的 MVC 实现的一致性，所以图 3-2 对总体了解 Swing MVC 的根本思想是有所帮助的。

与大多数轻量 Swing 组件一样，JSlider 维护对其模型的一个引用。Swing 模型由接口定义，滑杆的模型实现 BoundedRangeModel 接口。有边界范围的模型跟踪最小值、最大值和当前值^①。

Swing 提供缺省的模型实现，在组件模型没有被显式地指定时，则使用这个缺省模型。例如，如果一个滑杆的模型没有被显式地指定（通常都是这种情况），则用 DefaultBoundedRangeModel 的实例来实现滑杆。

所有的 Swing 轻量组件扩展 JComponent 类，该类维护一个对组件 UI 的引用。ComponentUI 类是 javax.swing.plaf 包中的一个抽象类，javax.swing.plaf 包定义 UI 代表的基本功能。

BasicSliderUI 类在 javax.swing.plaf.basic 包中并且封装了基本的按钮功能。标准 Swing 界面样式的滑杆 UI 类扩展 BasicSliderUI 类并定制了缺省功能。

BasicSliderUI 类实现六个内部类监听器，其中的五个监听 JSlider 组件；这个滑杆模型中还包含 BasicSliderUI.ChangeHandler。

3.2.3 动态认识

上一节提供了组成轻量 Swing 组件的对象之间关系的静态视图。本节提供一个组件的组成部分之间的相互关系的动态视图。

图 3-3 示出了一个图表，与图 3-1 的目的相似，它说明了 Swing 实现的 MVC 的信息流。

图 3-3 中示出的组件代表一个轻量 Swing 组件类，如 JButton、JLabel、JSlider 等等。

因为一个 UI 代表的监听器几乎总是作为内部类来实现的，所以，图 3-3 中的监听器包含在 UI 代表中。

图 3-3 中的模型代表组件的模型。例如，一个按钮的模型是 ButtonModel 接口的一个实现。

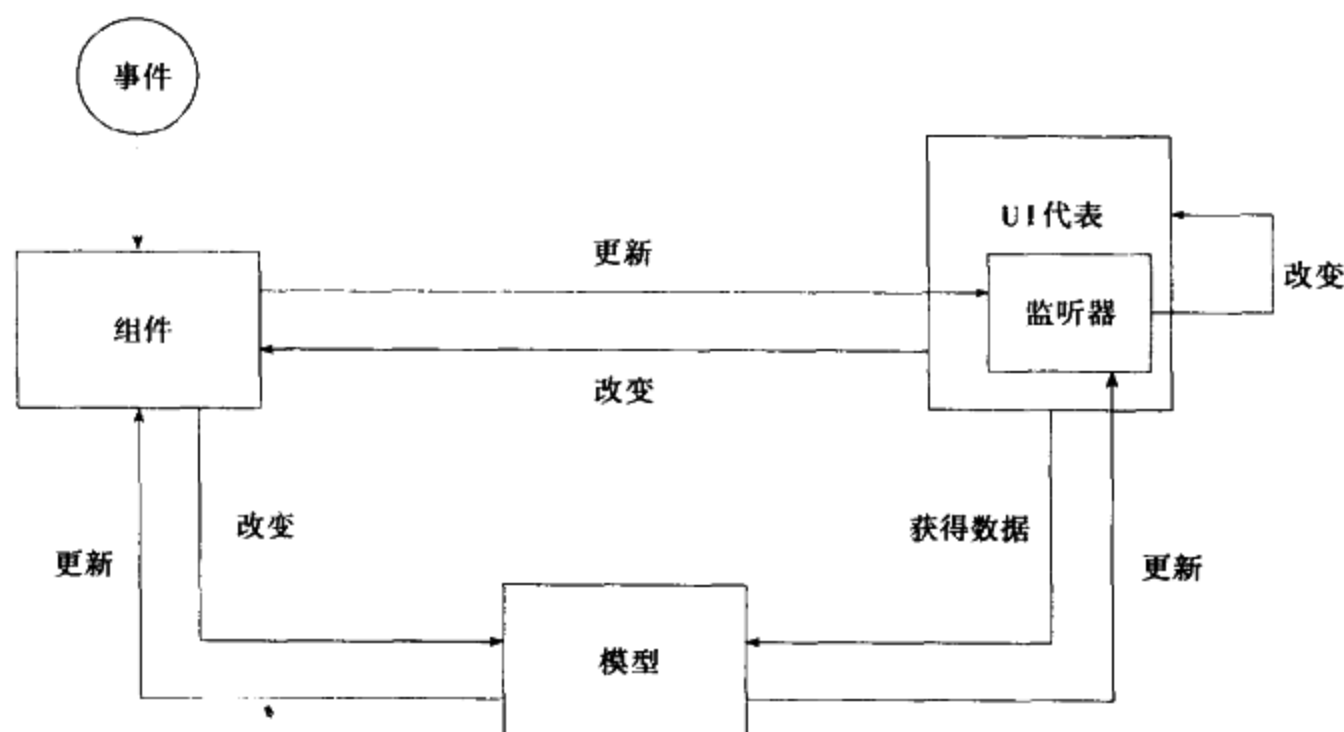


图 3-3 Swing MVC 通信

“Swing MVC”中曾作过介绍，组件为其模型提供传递方法，不用直接访问一个组件的模

① 有关滑杆和滑杆模型的更多信息，请参见 11.2 节“JSlider”

型就能操纵模型值。因此，图 3-3 示出了组件变化它们的模型。

我们还知道，JButton、JLabel 和 JSlider 等组件监听它们的模型，以便把模型事件传送给组件自己的监听器。因此，图 3-3 描述了这种模型，当模型变化时就更新组件。

UI 代表监听器通常监听组件或组件模型，所以，图 3-3 示出了被组件和模型更新的监听器。通常，监听器通过有选择地从模型获得信息并变化组件或 UI 代表来响应事件。

案例

图 3-3 图解说明了组成轻量 Swing 组件的对象相互通信的一般情况。本节介绍两个具体的例子以进一步阐明轻量 Swing 的组件通信。

图 3-4 示出了当一个滑杆属性被程序修改时所发生的事件序列。JSlider 的 paintTicks 属性是滑杆的 UI 代表的一个属性，不是一个模型属性。

Swing 滑杆维护一个 boolean 属性，该属性决定滑杆是否绘制勾号 (tick) 标记。图 3-4 示出了 JSlider.setPaintTicks() 被调用时发生的事件序列。

在设置了 paintTicks 属性后，JSlider.setPaintTicks() 调用 firePropertyChange()，firePropertyChange() 则向滑杆的属性变化监听器报告属性变化事件。滑杆属性变化监听器之一是 BasicSliderUI.PropertyChangeListener 的一个实例，它对这个事件的反映是强迫滑杆的 UI 代表更新滑杆。

在把变化情况通知给监听器之后，JSlider.setPaintTicks() 使滑杆重新生效并重画滑杆。

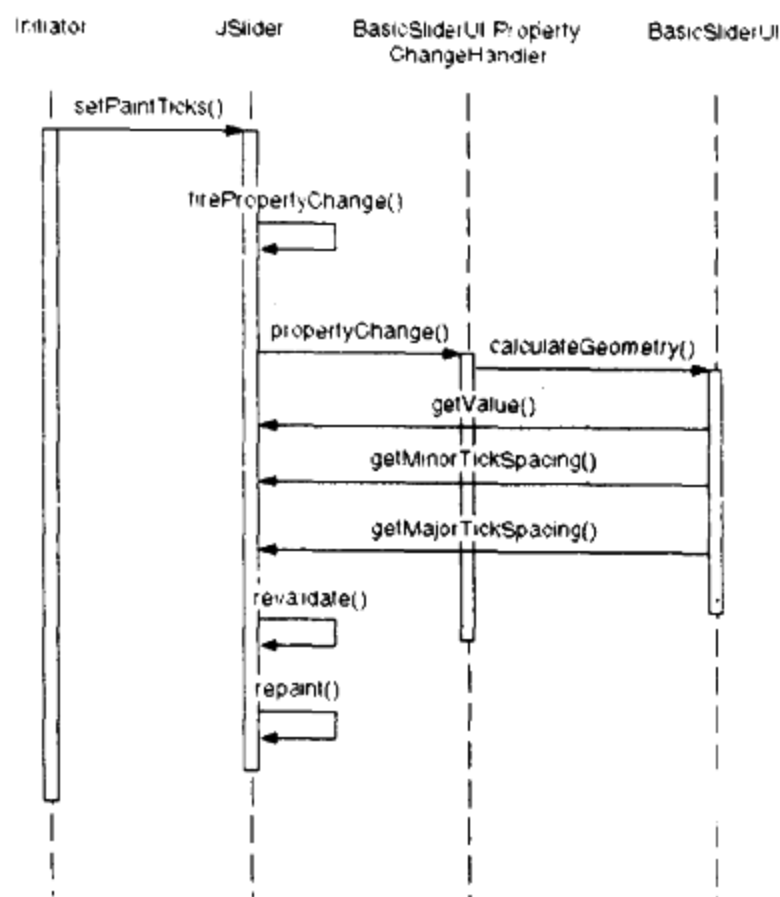


图 3-4 程序方式设置模型值

除了图 3-5 是为图 3-4 所示的事件序列所定制的以外，图 3-5 与图 3-3 类似。当 JSlider.setPaintTicks() 被激活后，这个滑杆更新 UI 代表的一个监听器。这个监听器便修改这个 UI 代表，这个 UI 代表获得这个组件本身的信息。

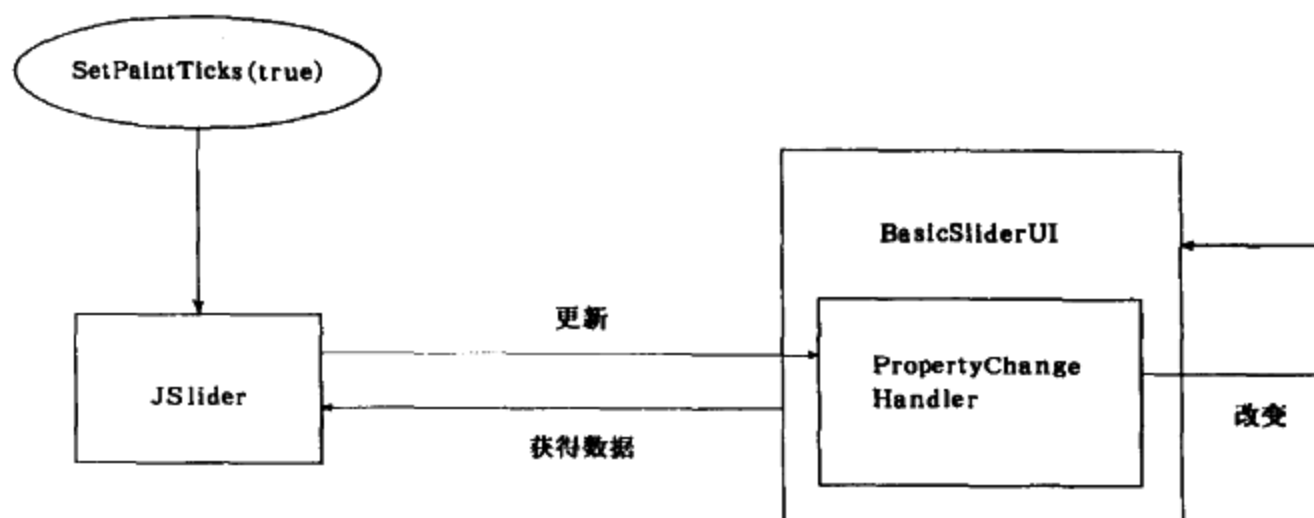


图 3-5 修改一个 JSlider 属性

图 3-6 示出了在滑杆的轨道上按下鼠标后发生的事件序列。鼠标按下事件派发给滑杆，滑杆响应该事件，把一个事件发送给它的监听器。它的监听器之一是 BasicSliderUI 的 TrackListener 的一个实例。

这个监听器通过调用 BasicSliderUI.scrollDueToClickInTrack()来操纵滑杆的 UI 代表。这个 UI 代表然后调用 JSlider.setValue()来设置滑杆的值。JSlider.setValue()方法是 DefaultBoundedRangeModel.setValue()的参数传递方法。参见 3.2.1 节“Swing 组件”中对模型属性的组件传递方法的讨论。

当这个模型值设置后，模型激发一个状态已变化事件，该事件由一个 BasicSliderUI.ChangeHandler 实例处理。变化监听器通过计算滑杆的滑块位置，然后重画滑杆来处理这个事件。

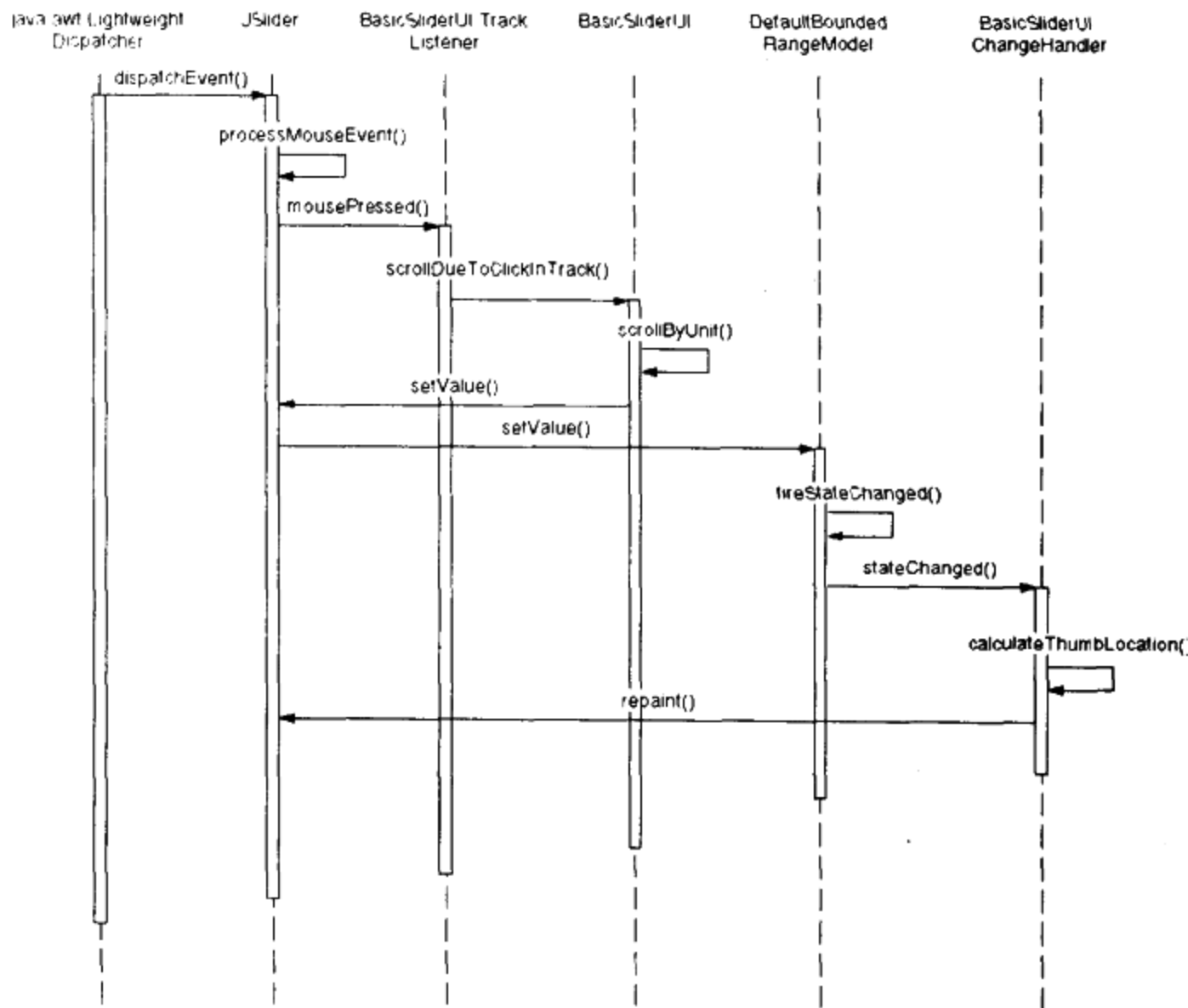


图 3-6 在滑杆轨道上按下鼠标的案例

图 3-7 示出了图 3-6 中说明的事件序列的组件通信。滑杆激发一个由轨道监听器处理的事件。该监听器修改 UI 代表（它更新滑杆），然后滑杆又更新模型。模型激发一个状态变化事件，这个事件由一个变化监听器通过重画组件来处理。

3.2.4 模型

大多数轻量 Swing 组件都有这样一个模型，这个模型维护状态信息，并在信息变化时激发

⊖ 有些组件（例如，JSeparator）没有模型

事件^①。例如，一个按钮的模型跟踪按钮的助记键及按钮是否待按下、按下或选取。按钮模型在它们的模型改变时将激发变化事件，当模型的选取状态变化时将激发项事件。

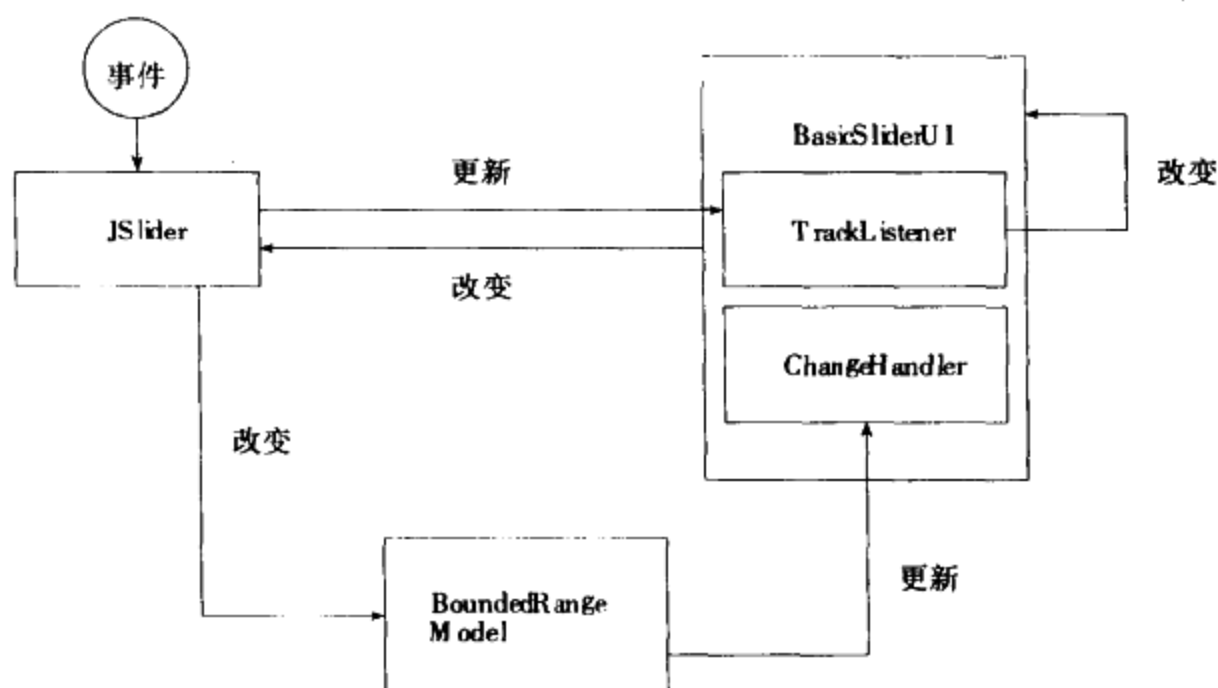


图 3-7 在滑杆的轨道中按下鼠标

Swing 提示

谁在监听？

Swing 组件由许多对象组成，例如，一个滑杆至少由九个对象组成，参见图 3-2 “滑杆组件类”。通过记住谁在监听谁有助于跟踪对象在做什么和了解对象之间是怎么交互的。

组件监听其模型，其主要目的是把事件传送给已向组件登记过的监听器。组件还为模型属性提供传递方法。传递模型事件和提供模型属性的传递方法减少了直接访问组件模型的需要。

UI 代表监听器主要监听组件，有时也直接监听组件的模型。UI 代表对组件和模型变化的响应通常是更新它们的外观，这通常要访问组件或模型，以便获得有关变化的更多信息。

模型是当作 JavaBeans 的关联属性来实现的。如果修改一个属性导致激发一个属性变化事件，则这个属性就是关联的，并且这个属性的访问方法遵循如下名字约定：

```
public void setModel (< ModelInterface > model)
public < ModelInterface > getModel ()
```

< ModelInterface > 表示定义模型类型的接口名。

1. 模型事件

模型具有激发大量事件的潜能。例如，当一个滑杆的滑块拖动时，该滑杆的模型激发一个连续不断的事件流，指示这个滑杆的值正在改变。因此，从性能方面来看，模型为每个激发的事件创建一个事件不总是实际的。

为了大大地减少由一个模型创建的事件对象的数量，模型激发一个由 javax.swing.event.ChangeEvent 类定义的特殊事件类型。变化事件与大多数其他事件不同，因为它们仅包含事件源这一种信息。这样，每个模型可以对所有的变化通知重复使用一个变化事件。

激发变化事件称为轻量通知，因为很少的信息与事件有关。激发其他的事件（例如，由按钮激发的动作事件）称为状态通知，因为该事件除包含事件源之外还包含许多状态信息。

轻量通知用于要经常修改的模型属性。监听轻量通知的监听器（指实现 ChangeListener 接口的监听器）询问从变化事件获得的事件源，以了解与变化有关的更多信息。

对很少变化的模型属性则使用状态通知。例如，从一个列表模型中删除一个元素产生一个状态通知，该通知包括删除行的索引值。

2. Swing 模型

表 3-1 列出了 Swing 模型接口以及与模型有关的组件。该表还指出了模型是否提供了轻量或状态通知以及 Swing 是否提供了模型接口的一个抽象实现。

表 3-1 Swing 模型

模型接口	使用者	通知 ^②	抽象类
BoundedRangeModel	JProgressBar、JScrollBar、JSlider	LW	
ButtonModel	JButton、JCheckBox、 JCheckBoxMenuItem、 JMenu、JMenuItem、 JRadioButton、JRadioButtonMenuItem JToggleButton	LW/ST	
ComboBoxModel	JComboBox	ST	
Document ^①	JEditorPane、JPasswordField、JTextArea、 JTextField、JTextPane	ST	
ListModel	JList	ST	
ListSelectionModel	JList、JTable	ST	
SingleSelectionModel	JMenuBar、JPopupMenu、JTabbedPane	LW	
TableModel	JTable	ST	
TableColumnModel	JTable	ST	
TreeModel	JTree	ST	
TreeSelectionModel	JTree	ST	

① Document 接口在 javax.swing.text 包中

② LW = 轻量通知，ST = 状态通知

所有的 Swing 模型都由 javax.swing 包中的接口定义，并且所有的 Swing 模型都有一个缺省的实现。例如，ButtonModel 接口由 DefaultButtonModel 类来实现，ListModel 由 DefaultListModel 来实现，如此类推。在没有为组件显式地设置模型时，就使用缺省实现。

有些模型如 ListModel、TableModel 和文本包中的 Document 接口，提供了抽象实现供开发人员去扩展。抽象的模型实现至少为监听器和事件激发方法提供了登记方法，这使得它们具有子类化的价值。提供抽象实现的模型是更复杂的 Swing 模型。

3. 一个模型的多个视图

MVC 体系结构的优点之一是可以把多个视图附加在单个模型上。而且，因为模型向它们的视图广播变化，所以模型的所有视图很容易保持同步。

图 3-8 所示的小应用程序含有两个组件：一个滑杆和一个滚动窗格，它们共享 DefaultBoundedRangeModel 的一个实例。滑杆使用这个模型来定位其滑块的位置，滚动窗格则使用这个模型来设置要显示的图形的比例。

本例中的滑杆和滚动窗格（ImageView 实例）共享一个 DefaultBoundedRangeModel 实例。通过调整滑杆来改变模型值会产生一个模型变化通知。滚动窗格响应这个模型通知，它根据这个模型值来调整它所显示的图像的比例。

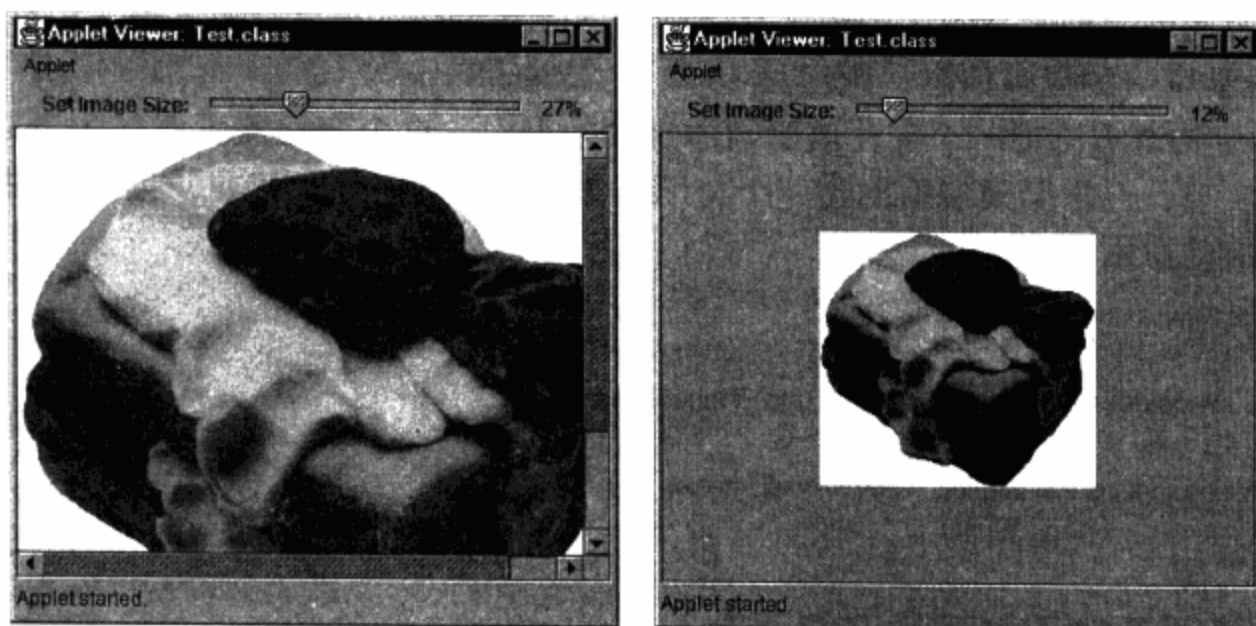


图 3-8 一个带多个视图的模型

本例的小应用程序创建模型、滑杆和滚动窗格。把这个模型传送给 JSlider 和 ImageView 的构造方法，并且还把一个变化监听器添加到模型中。

```
public class Test extends JApplet {
    DefaultBoundedRangeModel model =
        new DefaultBoundedRangeModel (100, 0, 0, 100);

    JSlider slider = new JSlider (model);
    JLabel readOut = new JLabel ("100%");

    ImageIcon image = new ImageIcon ("shortcake.jpg");
    ImageView imageView = new ImageView (image, model);

    Public void init () {
        Container contentPane = getContentPane ();
        JPanel panel = new JPanel ();

        panel.add (new JLabel (" Set Image Size:"));
        panel.add (slider);
        panel.add (readOut);

        contentPane.add (panel, BorderLayout.NORTH);
        contentPane.add (imageView, BorderLayout.CENTER);
        model.addChangeListener (new ReadOutSynchronizer ());
    }
    ...
}
```

这个小应用程序的变化监听器对模型变化作出反应，它更新显示图像比例的 readOut 标签。此后还调用了这个标签的 revalidate 方法，以便这个标签重新布局 and 重画。有关 revalidate 方法的更多信息，请参见 4.3.5 节“Validate、Invalidate 和 Revalidate 方法”。

```
...
class ReadoutSynchronizer implements ChangeListener {
    public void stateChanged (ChangeEvent e) {
        String s = Integer.toString (model.getValue ());
        readOut.setText (s + "%");
        readOut.revalidate ();
    }
}
```


接下来定义一个扩展 JScrollPane 的 ImageView 类，这个 ImageView 类必须以一个图像图标和一个有边界范围的模型为参数来构造。有关图标的更多信息，请参见第 5 章“边框、图标和动作”。有关滚动窗格的更多信息，则参见 13.2 节“JScrollPane”。

ImageView 构造方法把一个变化监听器添加到模型中。这个变化监听器作为 ImageView 的一个内部类来实现，它根据模型值创建一个与原图像成比例的实例。接着，把与原图像成比例的实例显示在滚动窗格上。

```

Class ImageView extends JScrollPane {
    ...
    public ImageView (ImageIcon icon, BoundedRangeModel model) {
        ...
        model.addChangeListener (new ModelListener ());
        ...
    }
    class ModelListener implements ChangeListener {
        public void stateChanged (ChangeEvent e) {
            BoundedRangeModel model =
                (BoundedRangeModel) e.getSource ();

            if ( ! model.getValueIsAdjusting () ) {
                int min = model.getMinimum (),
                    max = model.getMaximum (),
                    span = max-min,
                    value = model.getValue ();

                double multiplier = (double) value / (double) span;

                multiplier = multiplier == 0.0?
                    0.01 : multiplier;

                Image scaled = originalImage.getScaledInstance (
                    (int) (originalSize.width * multiplier),
                    (int) (originalSize.height * multiplier),
                    Image.SCALE_FAST);

                icon.setImage (scaled);
            }
        }
    }
}

```

例 3-1 列出了图 3-8 所示的小应用程序的完整代码。

例 3-1 一个带多个视图的模型

```

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    DefaultBoundedRangeModel model =
        new DefaultBoundedRangeModel (100, 0, 0, 100);

    JSlider slider = new JSlider (model);
    JLabel readOut = new JLabel (" 100%");

    ImageIcon image = new ImageIcon (" shortcake.jpg");
}

```

```

ImageView imageView = new ImageView (image, model);

public void init () {
    Container contentPane = getContentPane ();
    JPanel panel = new JPanel ();

    panel.add (new JLabel (" Set Image Size:"));
    panel.add (slider);
    panel.add (readOut);

    contentPane.add (panel, BorderLayout.NORTH);
    contentPane.add (imageView, BorderLayout.CENTER);

    model.addChangeListener (new ReadOutSynchronizer ());
}

class ReadOutSynchronizer implements ChangeListener {
    public void stateChanged (ChangeEvent e) {
        if (! model. getValuesAdjusting ()) {
            String s = Integer.toString (model.getValue ());
            readOut.setText (s + "%");
            readOut.revalidate ();
        }
    }
}

class ImageView extends JScrollPane {
    JPanel panel = new JPanel ();
    Dimension originalSize = new Dimension ();
    Image originalImage;
    ImageIcon icon;

    public ImageView (ImageIcon icon, BoundedRangeModel model) {
        panel.setLayout (new BorderLayout ());
        panel.add (new JLabel (icon));

        this.icon = icon;
        this.originalImage = icon.getImage ();

        setViewportView (panel);
        model.addChangeListener (new ModelListener ());

        originalSize.width = icon.getIconWidth ();
        originalSize.height = icon.getIconHeight ();
    }

    class ModelListener implements ChangeListener {
        public void stateChanged (ChangeEvent e) {
            BoundedRangeModel model =
                (BoundedRangeModel) e.getSource ();

            if (! model.getValuesAdjusting ()) {
                int min = model.getMinimum (),
                    max = model.getMaximum (),
                    span = max - min,
                    value = model.getValue ();

                double multiplier = (double) value / (double) span;

                multiplier = multiplier == 0.0 ?
                    0.01 : multiplier;

                Image scaled = originalImage.getScaledInstance (
                    (int) (originalSize.width * multiplier),

```

```

        (int) (originalSize.height * multiplier),
        Image.SCALE_FAST);

    icon.setImage (scaled);
    panel.revalidate ();
}
}
}

```

4. 轻量事件通知

由“模型”一节我们知道，模型能够提供轻量通知和状态通知两种通知。轻量通知使用一个只知道事件源的 `ChangeEvent`（变化事件），状态通知则使用提供有关变化的更多信息的事件。

变化事件由一些对象来处理，这些对象的类实现 `ChangeListener` 接口。接口总结 3-1 对 `ChangeListener` 接口进行了总结。

接口总结 3-1 `ChangeListener`

```
public abstract void stateChanged (ChangeEvent)
```

与大多数监听器一样，`ChangeListener` 接口只定义了一个方法。`StateChanged` 方法以 `ChangeEvent` 的一个实例作为参数。类总结 3-1 中介绍了 `ChangeEvent` 类。

类总结 3-1 `ChangeEvent`

扩展：`java.util.EventObject`

构造方法

```
public ChangeEvent (Object source)
```

`ChangeEvent` 类仅提供了一个构造方法，没有提供其他方法。`ChangeEvent` 构造方法以事件源作为参数。

图 3-9 所示的小应用程序通过监控滑杆的值来说明轻量通知。一个变化监听器添加到这个小应用程序的滑杆中以获得滑杆值并更新这个小应用程序的状态区。

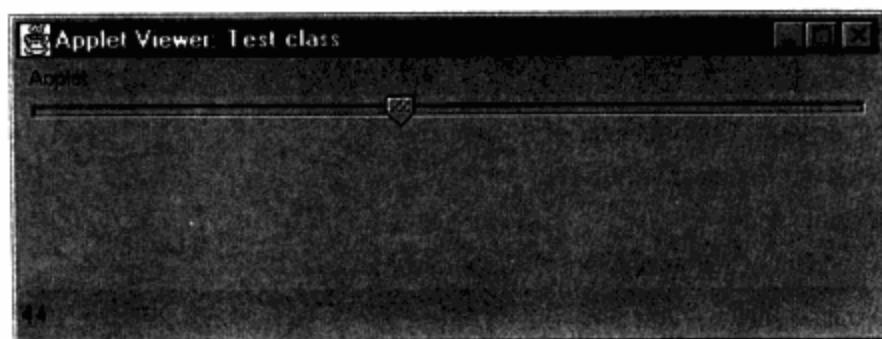


图 3-9 轻量通知

例 3-2 列出了图 3-9 所示的小应用程序的代码。

例 3-2 一个滑杆的轻量通知

```

import java.awt. * ;
import javax.swing. * ;
import javax.swing.event. * ;

public class Test extends JApplet {
    public void init () {

```

```

JSlider slider = new JSlider (0, 100, 50);
getContentPane ().add (slider, BorderLayout.CENTER);

slider.addChangeListener (new ChangeListener () {
    public void stateChanged (ChangeEvent e) {
        JSlider s = (JSlider) e.getSource ();
        showStatus (Integer.toString (s.getValue ()));
    }
});

```

5. 状态事件通知

对不经常变化的模型属性，模型使用状态通知。状态通知激发所有类型的事件，并且提供比事件源更多的信息（轻量事件通知仅提供事件源一种信息）。例如，当选取或取消选取一个单选按钮时，按钮模型将激发一个项事件。

6. 属性变化通知

当模型的关联属性变化时^①，模型会以一个 `java.beans.PropertyChangeEvent` 的形式产生状态通知。属性变化通知由一些对象来处理，这些对象的类实现 `java.beans.PropertyChangeListener` 接口，接口总结 3-2 中总结了这个接口。

接口总结 3-2 `PropertyChangeListener`

```
public void propertyChange (PropertyChangeEvent)
```

`PropertyChangeListener` 只定义了一个方法，该方法所带的参数是 `PropertyChangeEvent` 的一个实例。类总结 3-2 总结了 `PropertyChangeEvent` 类。

类总结 3-2 `PropertyChangeEvent`

扩展：`java.util.EventObject`

1. 构造方法

```
public PropertyChangeEvent (Object source, String propertyName,
                           Object oldValue, Object newValue)
```

属性变化事件以事件源、属性名、属性的旧值和新值为参数来构造。

2. 方法

```

public String getPropertyName ()
public Object getNewValue ()
public Object getOldValue ()
public void setPropagationId (Object propagationId)
public Object getPropagationId ()

```

属性变化监听器一般都要注意属性名的变化，因为大多数监听器对它们处理的属性是有选择的。同样，几乎所有的属性变化监听器都要跟踪属性的新值。属性变化监听器设计如下：

```

// code fragment...

SomePropertyChangeListener implements PropertyChangeListener {
    Public void propertyChange (PropertyChangeEvent e) {

```

① 当一个属性的变化激发一个属性变化事件，则这个属性称作关联属性。

```
String name = e.getName ();
// if property is one this listener is interested in ...
if ( name.equals ( " PropertyImInterestedIn" ) ) {
    SomeType newValue = (SomeType) e.getNewValue ();
    //act upon new value...
}
```

图 3-10 所示的应用程序用一个树和一个用于设置该树的 rootVisible 属性的复选框举例说明了处理模型属性变化事件。如果 rootVisible 属性是 true，则树的根节点是可视的，否则，根节点是隐藏的。

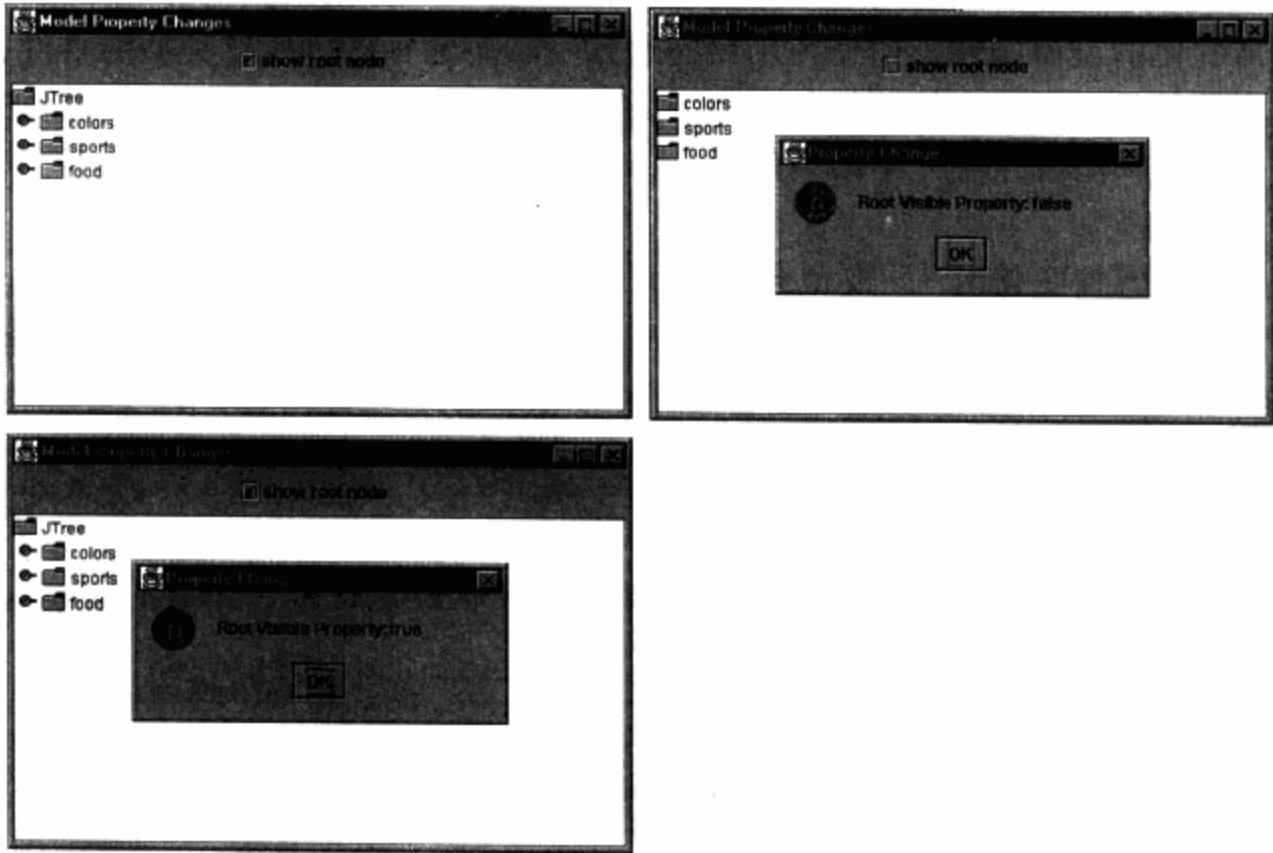


图 3-10 模型属性变化通知

这个应用程序实现一个属性变化监听器，该监听器在树的 rootVisible 属性修改时，显示一个消息对话框。

```
Class PropertyListener implements PropertyChangeListener {
    public void propertyChange (PropertyChangeEvent e) {
        String name = e.getPropertyName ();
        If ( name.equals ( JTree.ROOT_VISIBLE_PROPERTY ) ) {
            String msg = " Root Visible property: " +
                e.getNewValue () . toString ();
            JOptionPane.showMessageDialog (
                Test.this,           //parent comp
                msg,                 //message
                " Property Change",  //title
                JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

大多数属性变化监听器只对特定类型事件源的关联属性的一个子集感兴趣。属性变化监听器通常通过获取改变的属性的名字，并把它与一个公共常量相比较，以决定是否处理这个属性

变化。下面就是这种属性变化监听器的例子。

例 3-3 列出了图 3-10 所示的应用程序的完整代码。

例 3-3 处理模型属性变化通知

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.beans.*;

public class Test extends JFrame {
    JTree tree = new JTree ();

    public Test () {
        Container contentPane = getContentPane ();
        JScrollPane scrollPane = new JScrollPane (tree);

        contentPane.add (new ControlPanel (), BorderLayout.NORTH);
        contentPane.add (scrollPane, BorderLayout.CENTER);

        tree.addPropertyChangeListener (new PropertyListener ());
    }

    class ControlPanel extends JPanel {
        JCheckBox showRoot = new JCheckBox (" show root node");

        public ControlPanel () {
            showRoot.setSelected (tree.isRootVisible ());

            setLayout (new FlowLayout ());
            add (showRoot);

            showRoot.addActionListener (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    tree.setRootVisible (showRoot.isSelected ());
                }
            });
        }
    }

    class PropertyListener implements PropertyChangeListener {
        public void propertyChange (PropertyChangeEvent e) {
            String name = e.getPropertyName ();

            if (name.equals (JTree.ROOT_VISIBLE_PROPERTY)) {
                String msg = " Root Visible Property: " +
                    e.getNewValue ().toString ();

                JOptionPane.showMessageDialog (
                    Test.this,          // parent comp
                    msg,                // message
                    " Property Change", // title
                    JOptionPane.INFORMATION_MESSAGE);
            }
        }
    }

    public static void main (String args []) {
        GJApp.launch (new Test (),
            " Model Property Changes", 300, 300, 450, 300);
    }
}
```

Swing 提示**一个模型的多个视图**

一个 Swing 模型可以有多个视图，这点常常得不到人们的重视。初看起来，一个模型带多个视图是毫无意义的，例如，使一个进度条和一个滚动条共享一个模型有什么价值呢？

图 3-8 所示的小应用程序提供了一个模型带多个视图的例子。该小应用程序的滑杆和滚动窗格共享一个模型，以便滑杆的变化能在滚动窗格中显示的图像上得到反应。

还有许多其他的多个视图共享一个模型的情况。例如，可以在 JTable 实例和一个定制组件之间共享一个表格模型。这个定制组件绘制表格模型中所含数据的图表。把模型与视图分离使 Swing 组件更灵活和更具有再使用性。

3.2.5 UI 代表

组件把实现其用户界面 (UI) 的任务交给一个 UI 代表来完成。

UI 代表是在它们的组件的构造方法中实例化的，并且可以作为组件的一个关联属性来访问。例如，下面所列的 JSlider.java 的部分代码举例说明了一个滑杆是如何创建它的 UI 代表的以及滑杆的 UI 代表是如何被访问的。

```
//From JSlider.java;
//Note: the ui object below is a protected member of JComponent

class JSlider extends JComponent
    implements SwingConstants, Accessible {
    ...
    public JSlider (int orientation, int min, int max, int value) {
        ...
        updateUI ();
    }
    ...
    public void updateUI () {
        ...
        setUI ( (SliderUI) UIManager.getUI (this));
    }
    ...
    public SliderUI getUI () {
        return (SliderUI) ui;
    }
    ...
}
```

与其他的轻量 Swing 组件一样，JSlider 也通过 UIManager 获得 UI 代表。在 3.2.6 节中介绍了 UIManager.getUI() 实例化一个 UI 代表的方式；这里知道 UIManager.getUI() 返回一个 UI 代表就够了。

表 3-2 示出了在 JComponent 类中定义的、与组件的 UI 代表有关的方法[⊖]。

所有的轻量 Swing 组件都继承表 3-2 所列的方法，以设置和获得它们的 UI 代表，其中 up-

⊖ JComponent 类是所有轻量 Swing 组件的超类，，参见第 4 章“JComponent 类”。

dateUI 方法更新 UI 代表以便与当前的界面样式匹配。getUIClassID 方法返回一个表示组件的 UI 代表类的字符串，该方法还用于对 UI 代表实例化，参见 3.2.6 节中有关创建 UI 代表的介绍。

表 3-2 JComponent 的 UI 代表方法

方法 ^①	描述
<UI> getUI ()	返回对一个组件的 UI 代表的一个引用
void setUI (<UI>)	设置一个组件的 UI 代表
void updateUI ()	为当前的界面样式更新一个组件的 UI 代表
String getUIClassID ()	返回一个字符串，该字符串是这个 UI 代表的类名

① <UI> = UI 代表类名，例如，ButtonUI、LabelUI 等等。

表 3-2 中的 <UI> 指一个组件的 UI 代表的类名。例如，JButton 以如下方式定义表 3-2 所列的前两种方法：

```
ButtonUI getUI ()
void setUI (ButtonUI ui)
```

UI 代表的类名可以通过从组件的类名中去掉“J”并添加上一个“UI”来获得。例如，JButton 的 UI 代表的类名是 ButtonUI，JSlider 的 UI 代表类名则是 SliderUI。

UI 代表类名表示的是 javax.swing.plaf 包中的抽象类。例如，一个按钮的 UI 代表的全称类名是抽象的 javax.swing.plaf.ButtonUI。

javax.swing.plaf 包还包括抽象的 ComponentUI 类，该类为所有的 UI 代表定义基本行为。ComponentUI 类是 javax.swing.plaf 中抽象 UI 代表类的超类。例如，ButtonUI 和 SliderUI，以及所有其他的 javax.swing.plaf UI 类都扩展 javax.swing.plaf.ComponentUI。

javax.swing.plaf.basic 包提供实现 Swing 界面样式的通用部分的类。javax.swing.plaf.basic 包的 UI 代表提供缺省绘制功能和一些事件处理行为，它们可以在需要时用与界面样式有关的类来重载。

图 3-11 示出了 BasicSliderUI 类的类图。

滑杆的 UI 代表跟踪与滑杆的外观有关的属性。滑杆 UI 代表还为缺省事件处理行为使用了许多监听器。

BasicSliderUI 类被 MetalSliderUI、WindowsSliderUI 和 MotifSliderUI 等与界面样式有关的滑杆 UI 类扩展。

与“模型”中对模型的介绍一样，下面对 UI 代表的讨论将通过考察 UI 代表的案例，从静态认识转移到动态认识。

3.2.6 组件 UI 的案例

为了理解 UI 代表是怎样与它们的组件一起工作的，有必要了解两个关键案例：绘制 UI 代表，把 UI 代表安装到一个组件中。

1. 绘制 UI 代表

图 3-12 所示的流程图显示了在 Swing 按钮绘制时^①所发生的事件序列。图 3-12 中示出的序列对所有的轻量组件都是类似的。JButton.paintComponent() 为这个按钮的 UI 代表调用 update 方法。BasicButtonUI.paint() 方法绘制这个组件，这个按钮然后绘制该按钮的边框和这个组件的子组件。

如图 3-12 所示，JLabel、JButton 等组件类与它们的 UI 代表一起共同完成绘制任务。UI 代表负责绘制组件本身，组件类负责绘制组件的边框和组件的子组件^②。

① 有关绘制 Swing 轻量组件更详细的情况，请参见第 4.3 节“绘制 JComponents”。

② 大多数组件没有子组件。

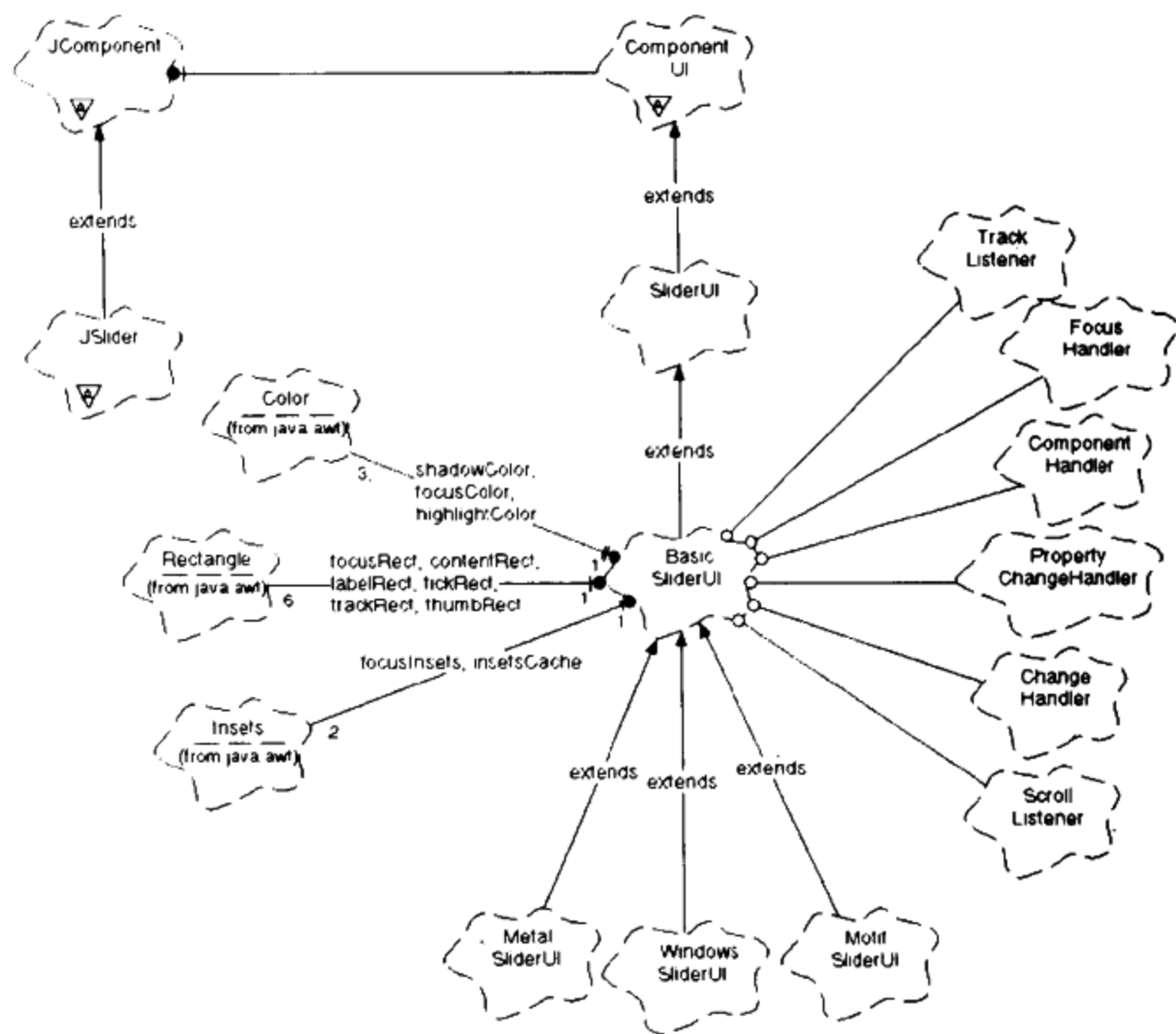


图 3-11 BasicSliderUI 类的类图

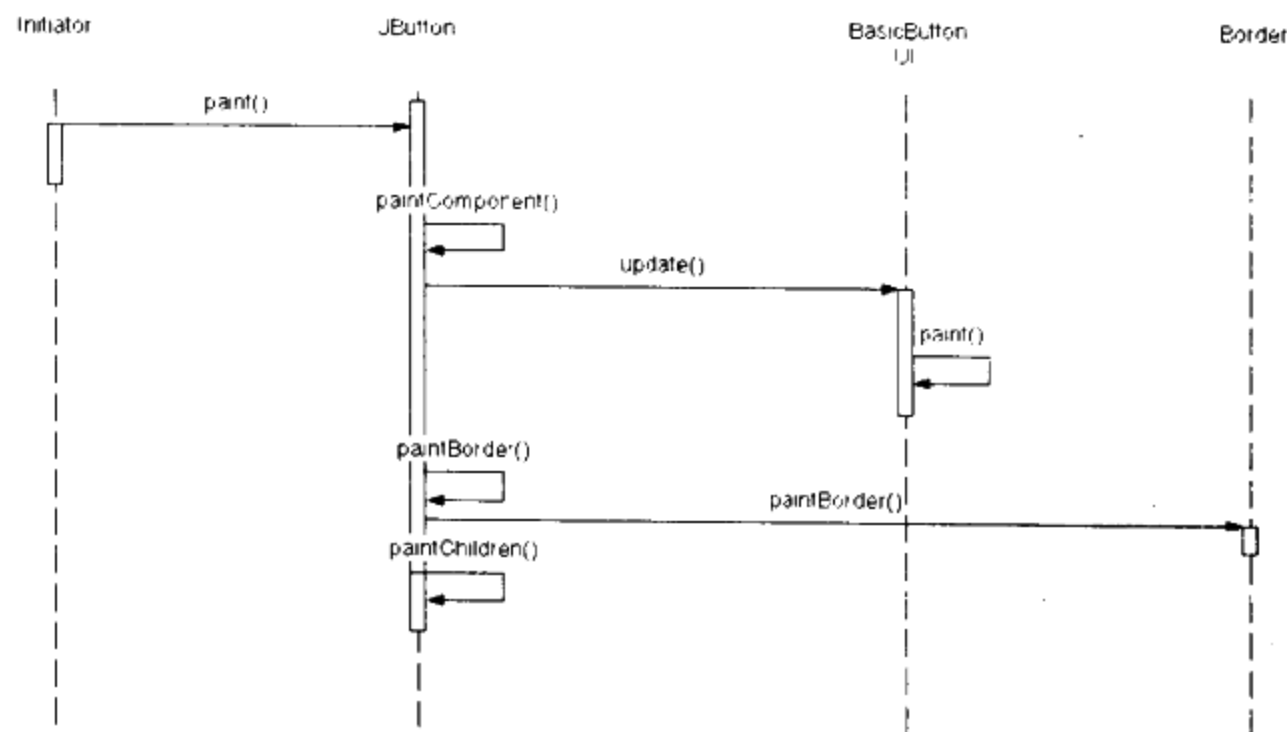


图 3-12 Swing 轻量组件的绘制

2. 安装一个 UI 代表

图 3-13 图解说明了把一个 UI 代表插入一个组件时所发生的事件序列。

从 `JButton` 构造方法中激活 `JButton.updateUI()`。与其他轻量 Swing 组件的 `updateUI` 方法一样，`JButton.updateUI()` 从 UI 管理器获得它的 UI 代表。这个按钮把一个对它本身的引用传送给

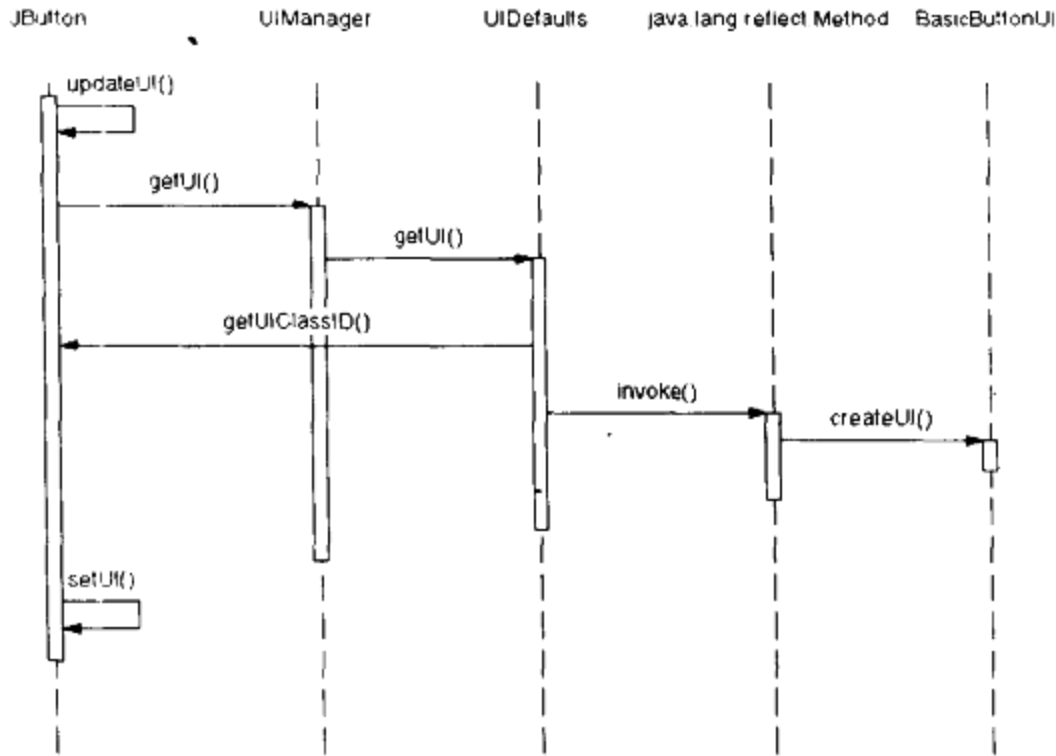


图 3-13 安装一个 UI 代表

UIManager.getUI() :

```
//From JButton.java:
public void updateUI () {
    setUI ( ( ButtonUI) UIManager.getUI (this));
}

```

UIManager.getUI()使用 UIDefaults 的一个实例，并调用 UIDefaults.getUI()来创建 UI 代表。

```
//From UIManager.java:
public static ComponentUI getUI (JComponent target) {
    ...
    ComponentUI ui = null;
    ...
    if (ui == null) {
        ui = getDefaults ().getUI (target);
    }
    return ui;
}

```

UIDefaults.getUI()是所有动作发生的地方，包括实例化 UI 代表。下面列出了 UIDefaults.getUI 方法：

```
// From UIDefaults.java:
public ComponentUI getUI (JComponent target) {
    ClassLoader uiClassLoader =
        target.getClass ().getClassLoader ();

    Class uiClass = getUIClass (target.getUIClassID (),
        uiClassLoader);

    object uiObject = null;
    if (uiClass == null) {
        getUIError (" no ComponentUI class for: " + target);
    }
    else {
        try {
            Method m = (Method) get (uiClass);

```



```

    if (m == null) {
        Class acClass = javax.swing.JComponent.class;
        m = uiClass.getMethod("createUI",
                               new Class[] {acClass});
        put(uiClass, m);
    }

    uiObject = m.invoke(null, new Object[] {target});
}

catch (NoSuchMethodException e) {
    getUIError("static createUI() method not " +
               "found in " + uiClass);
}

catch (Exception e) {
    getUIError("createUI() failed for " + target +
               " " + e);
}

return (ComponentUI) uiObject;
}

```

UIDefaults.getUI() 必须能够实例化任何类型的 UI 代表。例如，如果当前的界面样式是 Java 的界面样式，则必须为滑杆实例化 MetalSliderUI 实例；如果当前的界面样式是 Windows 的界面样式，则必须为滑杆实例化 WindowsSliderUI 实例，等等。另外，UIDefaults.getUI() 还必须能够实例化那些将开发出来的 UI 代表实例。显然，用 new 语句来实例化 UI 代表不是一个可行的选择。

为了实例化任何类型的 UI 代表，UIDefaults.getUI() 使用了 Java 的反射功能。UIDefaults.getUI() 以一个对组件的引用（UI 代表就是为这个组件而创建的）作为参数。再用这个组件来获得对这个 UI 代表类的引用。

有了 UI 代表类，再把反射功能用于获得对 UI 代表的 static createUI 方法的一个引用，然后再用 Method.invoke() 来调用这个方法（用这个组件作为参数）。所有 UI 代表实现一个 static createUI 方法，该方法返回一个对 UI 代表的引用。下面介绍有关 createUI 方法的更多信息。

1. ComponentUI 类

swing.plaf 包中的 ComponentUI 类是所有 Swing UI 代表的抽象基类。UI 代表不维护对组件的引用；反之，ComponentUI 类的方法接受一个对 JComponent 的引用作为参数。这样，一个 UI 代表可以被多个组件所共享。在 JComponent 上操作的共享对象是 Swing 的一个中心主题，例如，边框和图标设计成以一种类似的方式共享[○]。

ComponentUI 类存在于 swing.plaf 包中，并且实现下面的功能：

- 为一个组件安装和卸载 ComponentUI 类本身。
- 绘制和更新一个组件。
- 报告一个组件的首选大小、最小尺寸和最大尺寸。
- 报告一个点是否包含在一个组件内。
- 返回一个可以为多个组件所共享的 UI 实例。

类总结 3-3 列出了由 ComponentUI 类实现的 public 方法。

○ 参见第 5 章“边框、图标和动作”。

类总结 3-3 javax.swing.plaf. ComponentUI

扩展: java.lang.Object

1. 构造方法

```
public ComponentUI ()
```

因为 ComponentUI 不显式地实现任何构造方法, 所以, 这个唯一的构造方法由编译器产生 (因此有一个空实现)。

ComponentUI 的实例不能被实例化, 因为这个类是抽象的。因此, 只能从一个扩展类的构造方法中调用 swing.plaf. ComponentUI 的构造方法。

ComponentUI 的扩展通常通过 static createUI 方法实例化。

2. 方法

(1) UI 创建方法

```
public static ComponentUI createUI (JComponent)
```

为了促进共享和允许使用反射来实例化 UI 代表, 不要直接在它们的类之外实例化 UI 代表。而是通过调用 static createUI 方法来获得一个给定组件的 UI, 与下面 MotifButtonUI 所采用的方法类似:

```
// from swing.plaf.motif.MotifButtonUI;
private final static MotifButtonUI motifButtonUI =
    new MotifButtonUI ();
...
public static ComponentUI createUI (JComponent c) {
    return motifButtonUI;
}
```

如果单个 UI 代表能够被一个特定组件的所有实例所共享, 则 createUI 被重载以返回一个对单个 UI 实例的一个引用。例如, 具有 Motif 界面样式的所有 Swing 按钮共享 MotifButtonUI 的一个实例。结果, MotifButtonUI.createUI () 返回一个对 MotifButtonUI 的一个 static 实例的引用。

swing.plaf. ComponentUI. CreateUI () 弹出一个错误信息, 指出这个模型必须用 ComponentUI 类的扩展来实现。与这个错误有关的消息是: “ComponentUI . CreateUI 没有实现”。

(2) 安装/卸载

```
public void installUI (JComponent)
public void uninstallUI (JComponent)
```

当 UI 代表安装在一个组件上或从一个组件上卸载下来时, 经常要求 UI 代表执行一个或多个动作。通常, 这些动作包括添加或删除监听器或设置键盘动作。UI 代表是十分小心的, 以确保在安装 UI 代表时所发生的动作在卸载 UI 代表时不会进行, 以便组件保持在安装 UI 代表前的状态。

swing.plaf.ComponentUI 类的 InstallUI 和 uninstallUI 两个方法都是以无操作 (no-ops) 形式实现的。

(3) 绘制/更新/命中检测

```
public void paint (Graphics, JComponent)
public void update (Graphics, JComponent)
public boolean contains (JComponent, int, int)
```

UI 代表还负责绘制和更新组件。只有 ComponentUI 类的扩展才实现 paint 方法, 而 ComponentUI 版本的 paint () 什么也不做。如果组件是不透明的, 则 ComponentUI.update () 擦除组

件的背景。在处理完背景问题后，`update()` 调用 `paint()`。对 Swing 组件的绘制顺序的全面了解，请参见图 4-10 “`JComponent.paint()` 方法的调用顺序图”。

`contains` 方法返回一个 `boolean` 值，该值指出一个点是否在一个特定的组件边界内。缺省时，`contains` 方法为请求的组件返回 `JComponent.contains(int x, int y)` 的值。对那些外观不是矩形的组件，`ComponentUI.contains` 可以重载以考虑组件的形状。

(4) 组件大小

```
public Dimension getMaximumSize (JComponent)
public Dimension getMinimumSize (JComponent)
public Dimension getPreferredSize (JComponent)
```

为一个组件计算首选大小、最小尺寸、最大尺寸也是 `ComponentUI` 的责任。`ComponentUI.getPreferredSize()` 返回一个 `null` 大小，因此，很明显，必须通过扩展来重载 `ComponentUI.getPreferredSize()`。缺省时，最小尺寸和最大尺寸与首选大小相同，`ComponentUI.getMaximumSize` 和 `ComponentUI.getMinimumSize` 都返回 `ComponentUI.getPreferredSize()` 返回的值。`ComponentUI` 的扩展通常重载 `getMinimumSize()` 和 `getMaximumSize()`。

Swing 提示

一个 UI 代表的多个组件

与模型可以有多个视图一样，UI 代表也可以（而且经常是）有多个组件。每个在 `ComponentUI` 类中定义的公用方法都以一个对 `JComponent` 的引用作为参数，这意味着 `ComponentUI` 的扩展可以被多个组件所使用。

`ComponentUI` 扩展必须重载静态 `ComponentUI.createUI` 方法来返回对一个 UI 代表的引用。对能与多个组件一起使用的 UI 代表而言，每次调用 `createUI` 方法，这个方法都返回对 UI 代表的同一个引用。

2. 定制 UI 代表

实现用于特定组件或用于给定类型的所有组件的定制 UI 代表是十分简单的事情。

图 3-14 所示的小应用程序用 `PopOutButton` 定制 UI 代表，该代表指定为这个小应用程序按钮的 UI 代表。这个 UI 代表使用了较大的图标作为鼠标进入时按钮的图标及按钮按下时的图标。

图 3-14 中的左上图显示了光标将进入按钮时的小应用程序，右上图显示了光标刚进入按钮时的小应用程序。左下图显示了按钮按下时的小应用程序，右下图显示了按钮激活后的小应用程序。

这个小应用程序创建一个按钮并把该按钮的 UI 代表设置为 `PopOutButtonUI` 的一个实例。然后把这个按钮添加到小应用程序的内容窗格中。

```
public class Test extends JApplet {
    private String s = new String ();

    public void init () {
        Container contentPane = getContentPane ();
        JButton button = new JButton (new ImageIcon ("punch.gif"));
        button.setUI (new PopOutButtonUI ());
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);
    }
}
```

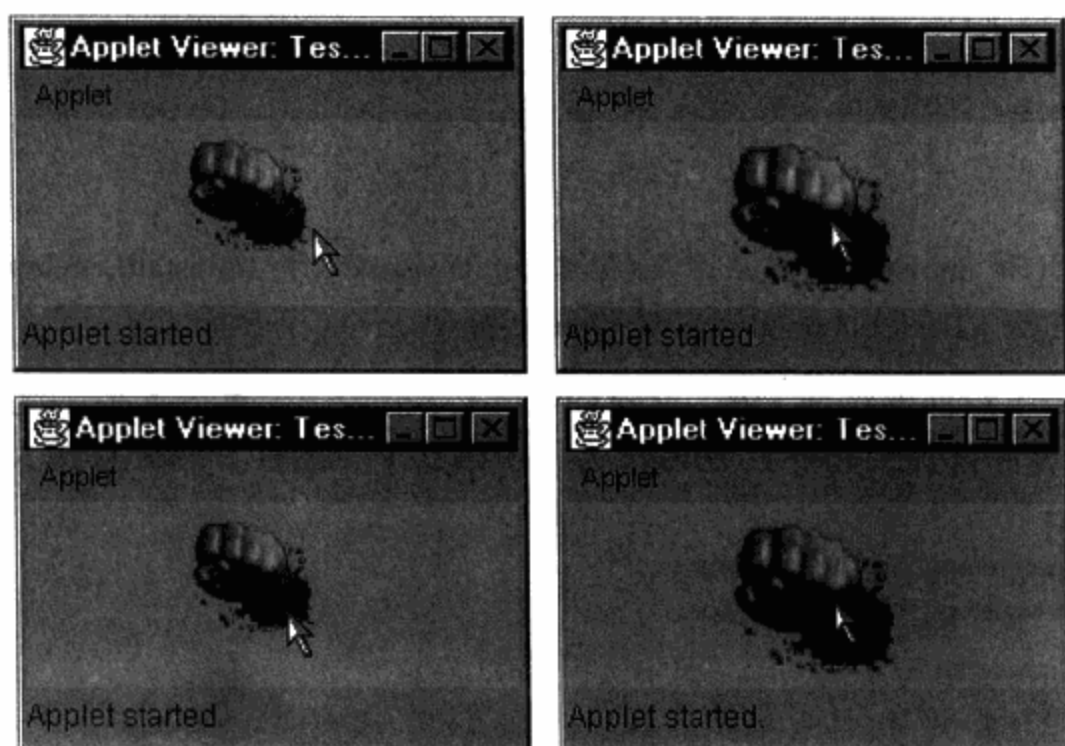


图 3-14 一个定制的界面样式

PopOutButtonUI 类扩展 BasicButtonUI 并重载下面的 ComponentUI 方法：

```
public void installUI (JComponent)
public void uninstallUI (JComponent)
public Dimension getPreferredSize ()
public boolean contains (JComponent, int x, int y)
public void paint (Graphics, JComponent)
```

installUI 方法获取这个按钮的边框，并把它作为组件的一个客户属性来保存，参见第 4.9 节“客户属性”中有关组件和客户属性的介绍。这个按钮的边框被作为客户属性保存，是因为 installUI 方法把这个按钮的边框设置为 null，因此，必须在 uninstallUI 方法中重新获得这个边框并恢复它。

InstallUI 方法还创建一个比按钮原来图标大 30% 的按钮图标。这个较大的图标被指定为鼠标进入这个按钮时的图标。

```
class PopOutButtonUI extends BasicButtonUI {
    public void installUI (JComponent c) {
        AbstractButton button = (AbstractButton) c;
        Border border = button.getBorder ();

        ImageIcon icon = (ImageIcon) button.getIcon ();
        int iconW = icon.getIconWidth ();
        int iconH = icon.getIconHeight ();

        Image scaled = icon.getImage ().getScaledInstance (
            iconW + (iconW/3),
            iconH + (iconH/3),
            Image.SCALE_SMOOTH);

        c.putClientProperty ("oldBorder", border);
        c.setBorder (null);

        button.setRolloverIcon (new ImageIcon (scaled));
        installListeners (button);
    }
}
```

```

public void uninstallUI (JComponent c) {
    Border border = (Border) c.getClientProperty (" oldBorder");
    c.putClientProperty (" oldBorder", null);
    c.setBorder (border);
    uninstallListeners (AbstractButton) c);
}
...

```

注意 installUI 和 uninstallUI 方法分别调用 installListeners 和 uninstallListeners, 以便安装或卸载由其超类 (BasicButtonUI) 实现的监听器。

UI 代表负责提供组件的首选大小。PopOutButtonUI 报告一个首选大小, 这个大小比 PopOutButtonUI 的超类计算的首选大小要大 30%。为了容纳这个大图标, 增大首选大小是必须的。

```

...
public Dimension getPreferredSize (JComponent c) {
    Dimension ps = super.getPreferredSize (c);
    ps.width += ps.width/3;
    ps.height += ps.height/3;
    return ps;
}
...

```

contains 方法被用于一个组件中的命中检测。PopOutButtonUI 重载 contains (), 以便只有当前显示的图标的边界内的点才被定义为被这个按钮所包容。

```

...
public boolean contains (JComponent c, int x, int y) {
    AbstractButton button = (AbstractButton) c;
    ButtonModel model = button.getModel ();
    Icon icon = getIcon (button, model);
    Rectangle iconBounds = new Rectangle (
        0, 0, icon.getIconWidth (), icon.getIconHeight ());
    return iconBounds.contains (x, y);
}
...

```

paint 方法被重载, 以便只绘制相应的图标。这个图标从一个 private getIcon 方法中获取, 如果按钮处于鼠标经过或按下状态, 则该方法返回按钮的大图标, 否则, 返回按钮的原图标。

```

...
public void paint (Graphics g, JComponent c) {
    AbstractButton button = (AbstractButton) c;
    ButtonModel model = button.getModel ();
    Icon icon = getIcon (button, model);
    Insets insets = c.getInsets ();
    icon.paintIcon (c, g, insets.left, insets.top);
}

private Icon getIcon (AbstractButton b, ButtonModel m) {
    return (m.isRollover () && ! m.isPressed ()) ?
        b.getRolloverIcon () : b.getIcon ();
}

```


例 3-4 列出了图 3-14 所示的小应用程序的完整代码。

例 3-4 一个定制的 UI 代表

```
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private String s = new String ();

    public void init () {
        Container contentPane = getContentPane ();
        JButton button = new JButton (new ImageIcon ("punch.gif"));
        button.setUI (new PopOutButtonUI ());
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                showStatus (s += ' ');
            }
        });
    }
}

class PopOutButtonUI extends BasicButtonUI {
    static protected BasicButtonListener listener;

    public void installUI (JComponent c) {
        AbstractButton button = (AbstractButton) c;
        Border border = button.getBorder ();

        ImageIcon icon = (ImageIcon) button.getIcon ();
        int iconW = icon.getIconWidth ();
        int iconH = icon.getIconHeight ();

        Image scaled = icon.getImage ().getScaledInstance (
            iconW + (iconW/3),
            iconH + (iconH/3),
            Image.SCALE_SMOOTH);

        c.putClientProperty ("oldBorder", border);
        c.setBorder (null);

        button.setRolloverIcon (new ImageIcon (scaled));
        installListeners (button);
    }

    public void uninstallUI (JComponent c) {
        Border border = (Border) c.getClientProperty ("oldBorder");

        c.putClientProperty ("oldBorder", null);
        c.setBorder (border);
        uninstallListeners ((AbstractButton) c);
    }

    public Dimension getPreferredSize (JComponent c) {
        Dimension ps = super.getPreferredSize (c);
    }
}
```

```

        ps.width += ps.width/3;
        ps.height += ps.height/3;
        return ps;
    }

    public boolean contains (JComponent c, int x, int y) {
        AbstractButton button = (AbstractButton) c;
        ButtonModel model = button.getModel ();
        Icon icon = getIcon (button, model);

        Rectangle iconBounds = new Rectangle (
            0, 0, icon.getIconWidth (), icon.getIconHeight ());

        return iconBounds.contains (x, y);
    }

    public void paint (Graphics g, JComponent c) {
        AbstractButton button = (AbstractButton) c;
        ButtonModel model = button.getModel ();

        Icon icon = getIcon (button, model);
        Insets insets = c.getInsets ();

        icon.paintIcon (c, g, insets.left, insets.top);
    }

    private Icon getIcon (AbstractButton b, ButtonModel m) {
        return (m.isRollover () && ! m.isPressed ()) ?
            b.getRolloverIcon () : b.getIcon ();
    }
}

```

3.2.7 监听器

如果一个轻量 Swing 组件要对事件作出反应（与所有的 Swing 组件所做的那样），则它应该有一个或多个作为该组件的 UI 代表的内部类来实现的监听器^①。

UI 代表通常实现 `protected installListeners` 和 `uninstallListeners` 方法，这两个方法分别从 `installUI` 和 `uninstallUI` 中调用。下面列出了 `BasicSliderUI.installListeners()` 和 `BasicSliderUI.UninstallListeners()` 方法：

```

// From javax.swing.plaf.basic.BasicSliderUI:

protected void installListeners (JSlider slider) {
    slider.addMouseListener (trackListener);
    slider.addMouseMotionListener (trackListener);
    slider.addFocusListener (focusListener);
    slider.addComponentListener (ComponentListener);
    slider.addPropertyChangeListener (propertyChangeListener);
    slider.getModel ().addChangeListener (ChangeListener);
}

...

protected void uninstallListeners (JSlider slider) {
    slider.removeMouseListener (trackListener);
    slider.removeMouseMotionListener (trackListener);
    slider.removeFocusListener (focusListener);
}

```

① 有 `BasicButtonListener` 类的按钮是这条规则的例外。

```

slider.removeComponentListener (componentListener);
slider.removePropertyChangeListener (
    propertyChangeListener);
slider.getModel ().removeChangeListener (changeListener);
}

```

缺省时，每个滑杆有六个监听器监听各种类型的事件。其中五个监听器监听滑杆，变化监听器则直接监听滑杆的模型。

定制监听器

因为轻量 Swing 组件有可插入的 UI 代表，所以一个组件的事件处理也是可插入的。

大多数扩展 `javax.swing.plaf.basic` 包中类的 UI 代表把监听器作为内部类来实现。作为一个 UI 代表的内部类来实现的事件监听器应该只在 UI 代表类中或在 UI 代表类的扩展中被实例化^①。因此，实现一个定制监听器意味着实现一个定制 UI 代表。

图 3-15 所示的小应用程序包括一个带有一个定制 UI 代表的滑杆——一个 `AnnotatedSliderUI` 实例。

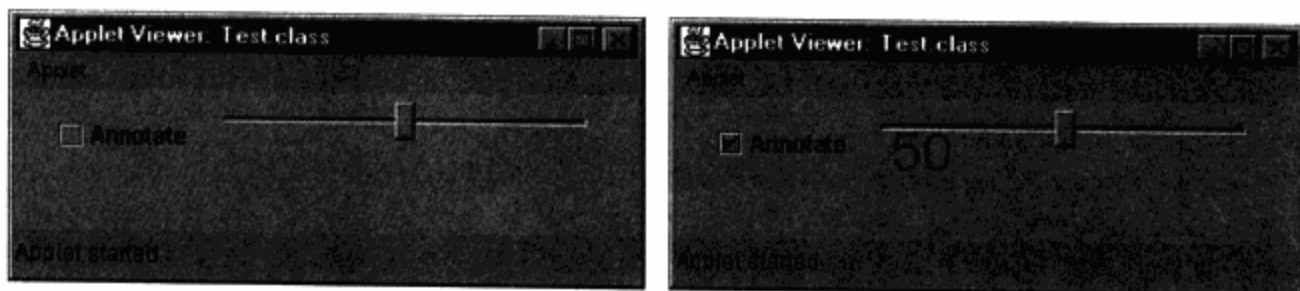


图 3-15 定制的事件的处理

如果这个滑杆有一个名为 `AnnotatedSliderUI.annotate` 的客户属性且其值为 `true`，则 UI 代表用滑杆的值来注释滑杆。如果滑杆没有注释客户属性或该属性的值是 `false`，则 UI 代表不注释这个滑杆。图 3-15 所示的小应用程序提供一个复选框来设置滑杆的注释客户属性^②。

这个小应用程序创建一个滑杆和一个复选框，并设置滑杆的 UI 代表为一个 `AnnotatedSliderUI` 实例。

动作监听器被添加到复选框中，以便根据检查框的选取状态来设置滑杆的 `annotate` 属性。在更新滑杆的 `annotate` 属性后，这个复选框的动作监听器重绘滑杆以反应这种变化。

```

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        final JSlider slider = new JSlider ();
        final JCheckBox checkBox = new JCheckBox (" Annotate");
        slider.setUI (new AnnotatedSliderUI (slider));
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (checkBox);
        contentPane.add (slider);

        checkBox.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                boolean selected = checkBox.isSelected ();
                slider.putClientProperty (

```

① 一个编译错误，迫使类是 `public` 类型，但是这些类应该是 `protected` 类型。

② 参见第 4.9 节“客户属性”中有关组件和客户属性的介绍。

```

        AnnotatedSliderUI.ANNOTATE_PROPERTY,
        selected ? Boolean.TRUE : Boolean.FALSE);

        slider.repaint ();
    }
}

```

AnnotatedSliderUI 类扩展 BasicSliderUI，并为滑杆的 `annotate` 属性的关键字提供一个 `public static` 字符串。这个字符串被复选框的动作监听器用来设置 `annotate` 属性。

AnnotatedSliderUI 重载 BasicSliderUI 的 `createChangeListener` 方法以返回 `AnnotateChangeListener` 的一个实例。通过重载 `createChangeListener()`，AnnotatedSliderUI 用自己的属性变化监听器来替代 BasicSliderUI 的属性变化监听器。

```

class AnnotatedSliderUI extends BasicSliderUI {
    public static String ANNOTATE_PROPERTY =
        "AnnotatedSliderUI.annotate";

    JSlider s;
    boolean annotate = false;
    ...
    protected PropertyChangeListener
        createChangeListener (JSlider slider) {
        return new AnnotateChangeListener ();
    }
    ...
}

```

AnnotatedSliderUI.`PropertyChange()` 调用其超类的 `PropertyChange` 方法 (BasicSliderUI.`PropertyChangeHandler`) 以确保以缺省方式来处理其他滑杆关联属性的变化。

有关处理属性变化事件的更多信息，请参见 3.2.3 节。

```

...
public class AnnotateChangeListener
    extends BasicSliderUI.PropertyChangeHandler {
    public void propertyChange (PropertyChangeEvent e) {
        super.propertyChange (e);

        String name = e.getPropertyName ();
        if (name.equals (ANNOTATE_PROPERTY)) {
            if (e.getNewValue () != null) {
                annotate =
                    ((Boolean) e.getNewValue ()).booleanValue ();
            }
        }
    }
    ...
}

```

为了容纳注释，AnnotatedSliderUI 重载了 `getPreferredSize()`。

如果 `annotate` `boolean` 成员变量的值为 `true`，则 `paint` 方法绘制注释，然后调用 `super.paint()` 来绘制滑杆的其他部分。

```

...
public Dimension getPreferredSize (JComponent c) {
    Dimension d = super.getPreferredSize (c);
    return new Dimension (d.width, d.height + 20);
}

```

```

|
public void paint (Graphics g, JComponent c) {
    if (annotate) {
        JSlider slider = (JSlider) c;
        int v = slider.getValue ();

        g.setColor (UIManager.getColor (" Label.foreground"));
        g.setFont (new Font (" Dialog", Font.PLAIN, 28));
        g.drawString ( (new Integer (v)) .toString (), 10, 33);
    }
    super.paint (g, c);
}
...

```

注意，为了简单，`paint` 和 `getPreferredSize` 方法使用了能够产生所需效果的常量。一般不建议使用硬编码的常量。

例 3-5 列出了图 3-15 所示的小应用程序的完整代码。

例 3-5 一个定制监听器的实现

```

import javax.swing.*.*;
import javax.swing.event.*;
import javax.swing.plaf.basic.*;
import java.awt.*.*;
import java.awt.event.*;
import java.beans.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        final JSlider slider = new JSlider ();
        final JCheckBox checkBox = new JCheckBox (" Annotate");

        slider.setUI (new AnnotatedSliderUI (slider));

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (checkBox);
        contentPane.add (slider);

        checkBox.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                boolean selected = checkBox.isSelected ();

                slider.putClientProperty (
                    AnnotatedSliderUI.ANNOTATE_PROPERTY,
                    selected ? Boolean.TRUE : Boolean.FALSE);

                slider.repaint ();
            }
        });
    }
}

class AnnotatedSliderUI extends BasicSliderUI {
    public static String ANNOTATE_PROPERTY =
        " AnnotatedSliderUI.annotate";

    boolean annotate = false;

    public AnnotatedSliderUI (JSlider slider) {
        super (slider);
    }
}

```



```

public Dimension getPreferredSize (JComponent c) {
    Dimension d = super.getPreferredSize (c);
    return new Dimension (d.width, d.height + 20);
}

public void paint (Graphics g, JComponent c) {
    if (annotate) {
        JSlider slider = (JSlider) c;
        int v = slider.getValue ();

        g.setColor (UIManager.getColor (" Label.foreground"));
        g.setFont (new Font (" Dialog", Font.PLAIN, 28));
        g.drawString ( (new Integer (v)) .toString (), 10, 33);
    }

    super.paint (g, c);
}

protected PropertyChangeListener
    createPropertyChangeListener (JSlider slider) {
    return new AnnotatePropertyListener ();
}

protected class AnnotatePropertyListener
    extends BasicSliderUI.PropertyChangeListener {
    public void propertyChange (PropertyChangeEvent e) {
        super.propertyChange (e);

        String name = e.getPropertyName ();
        if (name.equals (ANNOTATE _ PROPERTY)) {
            if (e.getNewValue () != null) {
                annotate =
                    ((Boolean) e.getNewValue ()) .booleanValue ();
            }
        }
    }
}

```

3.3 本章回顾

Swing 建立在一个改进的模型-视图-控制器 (MVC) 体系结构上。MVC 体系结构是一个经过验证的设计, Smalltalk 应用程序的基础就是 MVC。MVC 把模型、视图和控制器封装为独立的对象, 这些对象通过一个特定的协议相互通信。通过封装其他应用程序的优点, MVC 应用程序 (和 Swing 组件) 比传统应用程序更具灵活性和再使用性。

第 4 章 JComponent 类

JComponent 类是所有 Swing 轻量组件的基类，因此，我们单独用一章对它进行讨论。JComponent 对 Swing 的意义就如同 `java.awt.Component` 对 AWT 的意义一样，它们都是它们各自框架组件的基类。

作为所有 Swing 轻量组件的基类，JComponent 提供了大量的基本功能。要全面了解 Swing，就必须知道 JComponent 类提供的功能，还必须知道如何使用 JComponent 类。

4.1 JComponent 类概览

JComponent 扩展 `java.awt.Container`，而 `java.awt.Container` 又扩展 `java.awt.Component`，因此，所有的 Swing 组件都是 AWT 容器。`Component` 和 `Container` 类本身提供了大量的功能，因此，JComponent 继承了大量的功能。本章（实际上本书）假设读者有 AWT 组件和容器的基本知识，这些基本知识在《Java2 图形设计，卷 I: AWT》中用了大量的篇幅来介绍。

因为 JComponent 为几乎所有的 Swing 组件提供下层构件，因此，它是一个很大的类，包括 100 多个 `public` 方法。JComponent 类为它的扩展提供了下面的功能：

- 边框。
- 可访问性。
- 双缓存。
- 调试图形。
- 自动滚动。
- 工具提示。
- 键击处理。
- 客户属性。

4.1.1 边框

任何 JComponent 的扩展都可以带边框。Swing 提供了许多不同风格的边框，如雕刻边框、带标题边框和蚀刻边框。虽然一个组件只能有一个边框，但是边框是可以组合的。因此，从效果上来看，单个组件可有多个边框。图 4-1 示出了组合边框、带标题边框和定制边框。

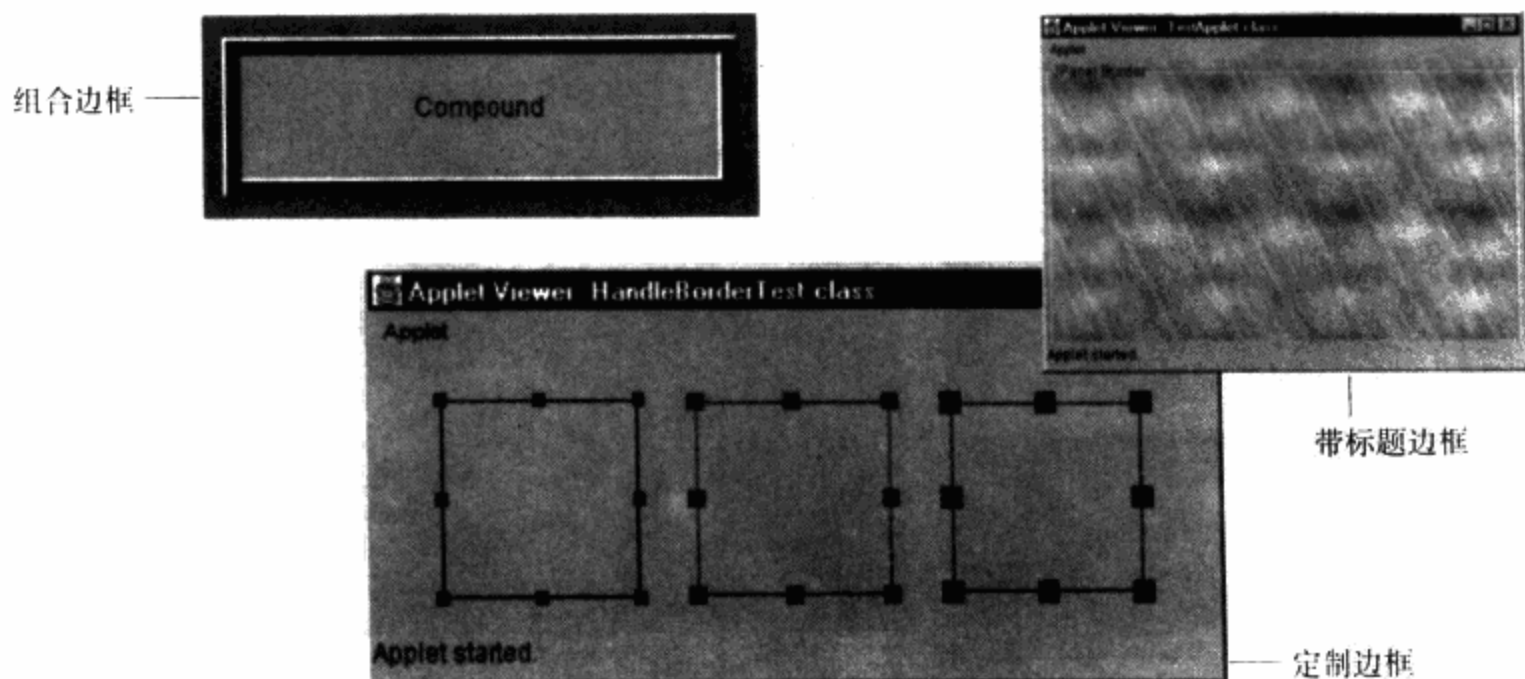


图 4-1 边框

边框通常用来组织组件集，但在其他情况下也是很有用的。例如，图 4-1 示出的组合边框可以作为一个显示艺术品略图的图形程序的图片帧。通常，可操作的边框在绘图程序中用来移动和改变对象的大小，而且作为 Swing 的定制边框，这种边框实现起来也很容易。

本章不详细介绍边框，有关边框的知识，请参见第 5 章“边框、图标和动作”。

4.1.2 可访问性

可访问性是使人人都能使用软件。例如，为视力不好的用户放大字体或为听力不好的用户显示带声音的标题。

Swing 的插入式界面样式体系结构通过允许把可选择的界面样式分配给一组组件来支持可访问性。图 4-2 所示的 SwingSet 样例应用程序使用一个定制的界面样式，它具有高反差、大字体外观，为视力不好的用户提供了更好的可读性。

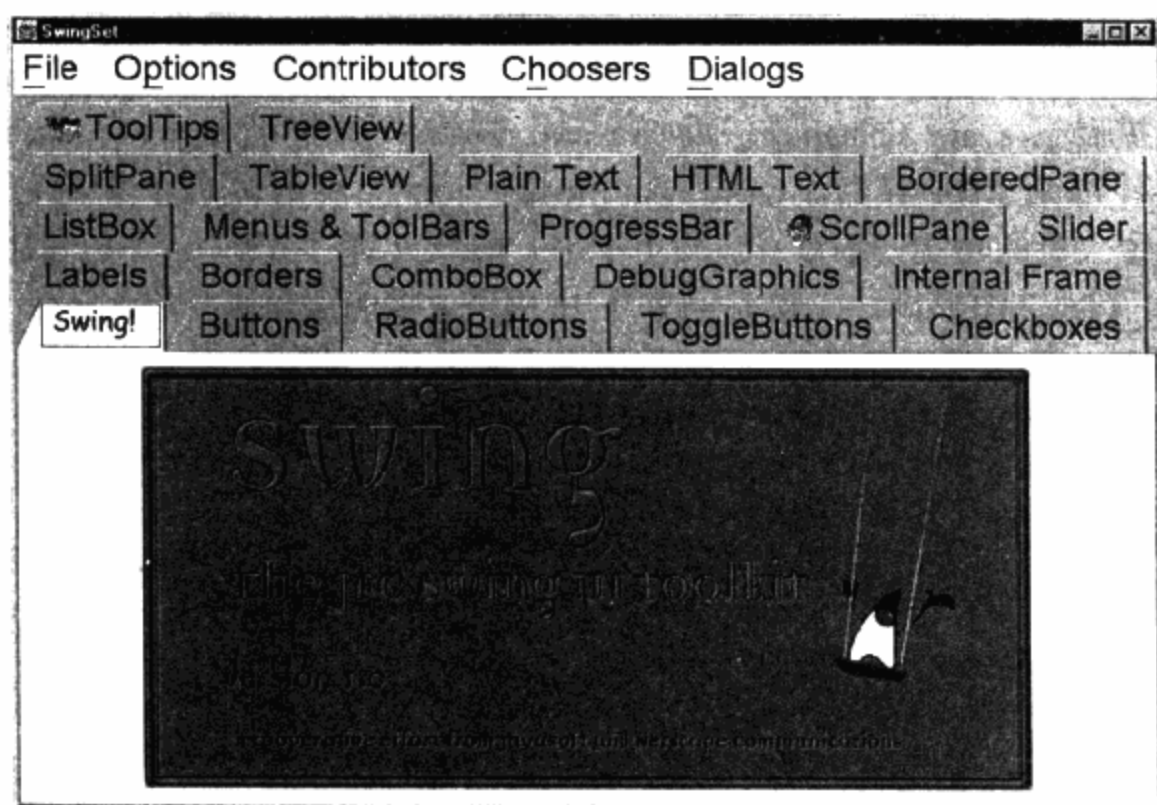


图 4-2 用定制的界面样式来支持可访问性

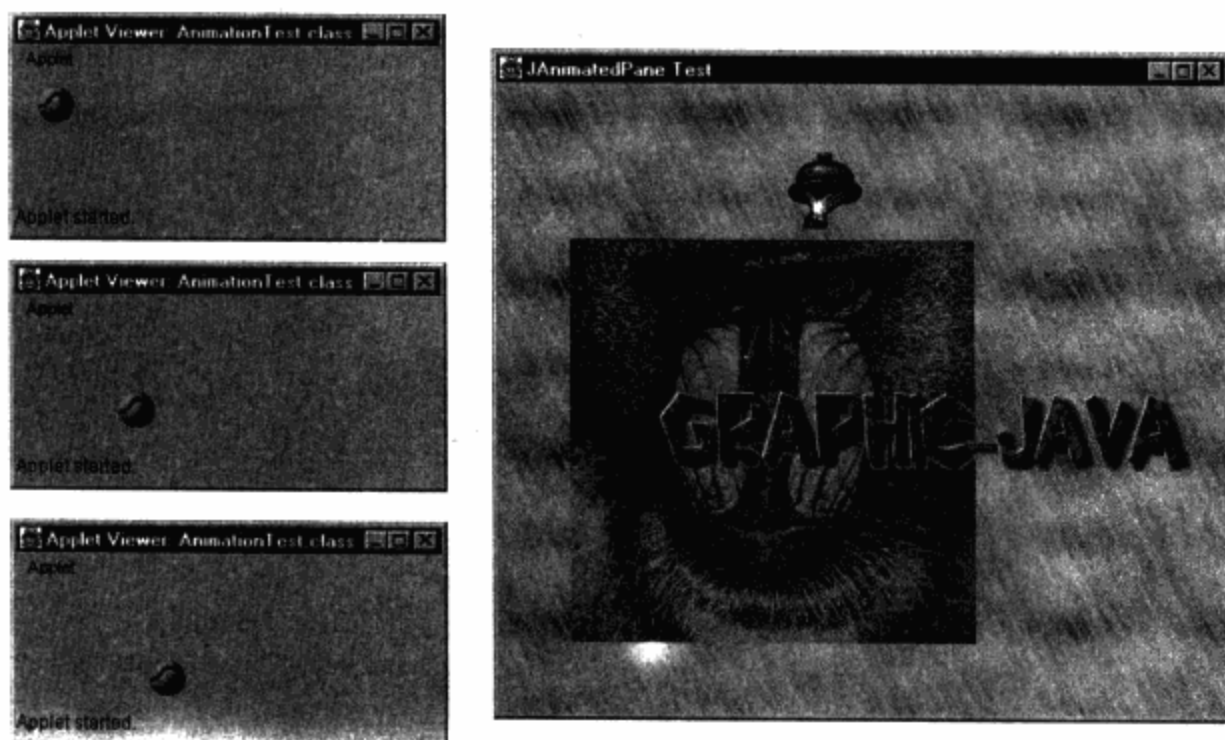


图 4-3 可拖动图标和动画使用的双缓存

除了 Swing 插入式界面样式外，使用一个可访问 API 和一组可访问工具也能支持可访问性。在第 4.11 节“支持可访问性”中介绍了可访问性。

4.1.3 双缓存

在更新组件（擦除然后重绘组件）时，会产生可察觉的闪烁。双缓存通过在屏外缓存区中更新组件，然后把屏外缓存区的相应部分拷贝到组件的屏上代表中来消除闪烁。

所有的 Swing 轻量组件都继承了双缓存它们显示内容的能力。一个屏外缓存（由 Swing 的 RepaintManger 维护）常用于双缓存 JComponent 的扩展。

图 4-3 示出了一个篮球图像的简单动画和含有可拖动的轻量组件的应用程序。

除了为双缓存轻量组件使用屏外缓存外，开发人员还可以为拖动轻量组件或实现动画等其他目的而使用屏外缓存。

4.1.4 调试图形

调用 JComponent.setDebugGraphicsOptions(int) 可以把一个组件转换成一个慢速绘制器，该绘制器在每个图形调用前闪烁一下，而且还可以维护一个图形调用的日志文件。

调试图形允许开发人员快速、准确地了解组件是如何绘制的，在实现一个需要大量绘制的定制组件时，调试图形是很方便的。图 4-4 显示了一个滑杆，它选取了调试图形闪烁选项。滑杆以很慢的速度绘制，在绘制之前用红色闪烁来指示每次的图形绘制。

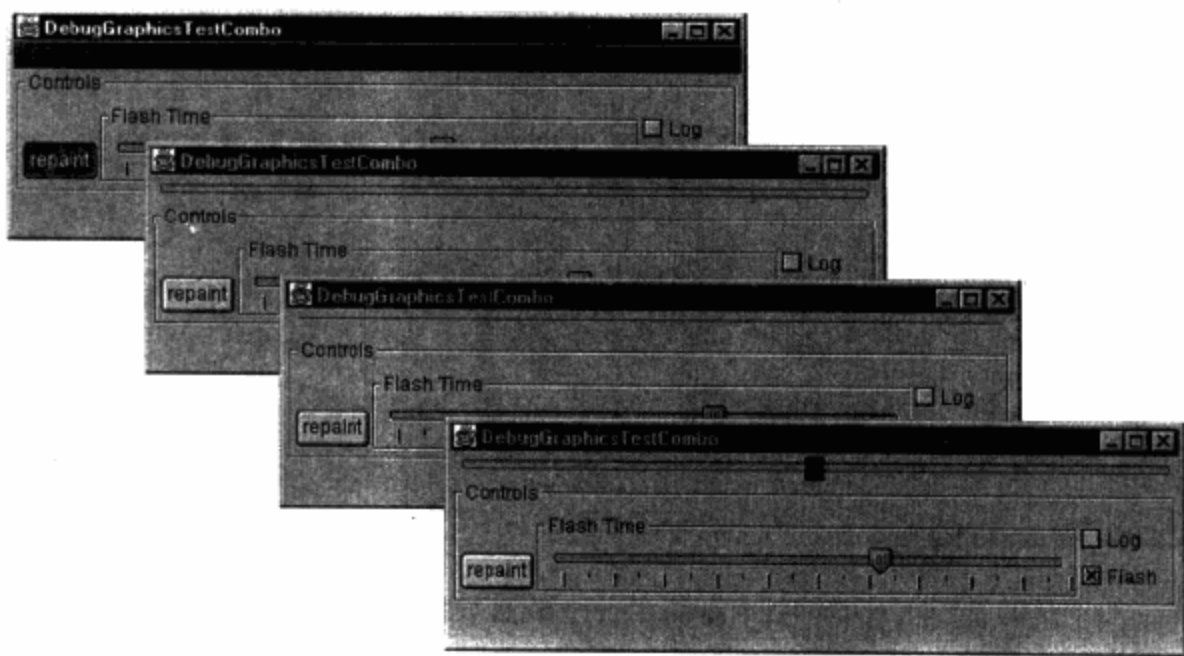


图 4-4 调试图形——慢速绘制

下面给出了与图 4-4 描述的图形操作有关的图形日志的一部分：

Graphics (0-3) Enabling debug

Graphics (0-3) Setting color:

Javax.swing.plaf.ColorUIResource [r = 153, g = 153, b = 204]

Graphics (0-3) Setting font:

Javax.swing.plaf.FontUIResource [family = Dialog, name = Dialog, style = plain, size = 12]

Graphics (1-3) Setting color:

Javax.swing.plaf.ColorUIResource [r = 204, g = 204, b = 204]

Graphics (1-3) Filling rect:

Javax.awt.Rectangle [x = 0, y = 0, width = 300, height = 21]

```
Graphics (1-3) Translating by: java.awt.Point [x=0, y=0]
Graphics (1-3) setting color:
Javax.swing.plaf.ColorUIResource [r=153, g=153, b=153]
Graphics (1-3) Drawing line: from (7, 8) to (292, 8)
Graphics (1-3) Drawing line: from (293, 8) to (293, 14)
Graphics (1-3) Drawing line: from (293, 15) to (8, 15)
Graphics (1-3) Drawing line: from (7, 15) to (7, 9)
Graphics (1-3) Translating by: java.awt.Point [x=0, y=0]
Graphics (1-3) Translating by: java.awt.Point [x=143, y=5]
Graphics (1-3) Translating by: java.awt.Point [x=0, y=0]
...
```

调试图形的很多方面都是可设置的，其中包括闪烁的颜色、闪烁的时间间隔和日志流等。

4.1.5 自动滚动

当把光标拖出了组件的边界时，Swing 轻量组件可以滚动其内容，这个特性称作自动滚动。所有的 JComponent 扩展都继承了自动滚动的功能。可以在每个组件的基础上设置允许或禁止自动滚动。缺省时，JList 和 JTable 是允许自动滚动的。

为了取得一些有趣的效果，还可在定制组件中实现自动滚动。例如，图 4-5 示出了一个定制的、能自动滚动的视口。

拖动图像本身会滚动图 4-5 视口中显示的图像。如果拖动时把光标拖出了视口的边界，则图像向光标移动的方向自动滚动。

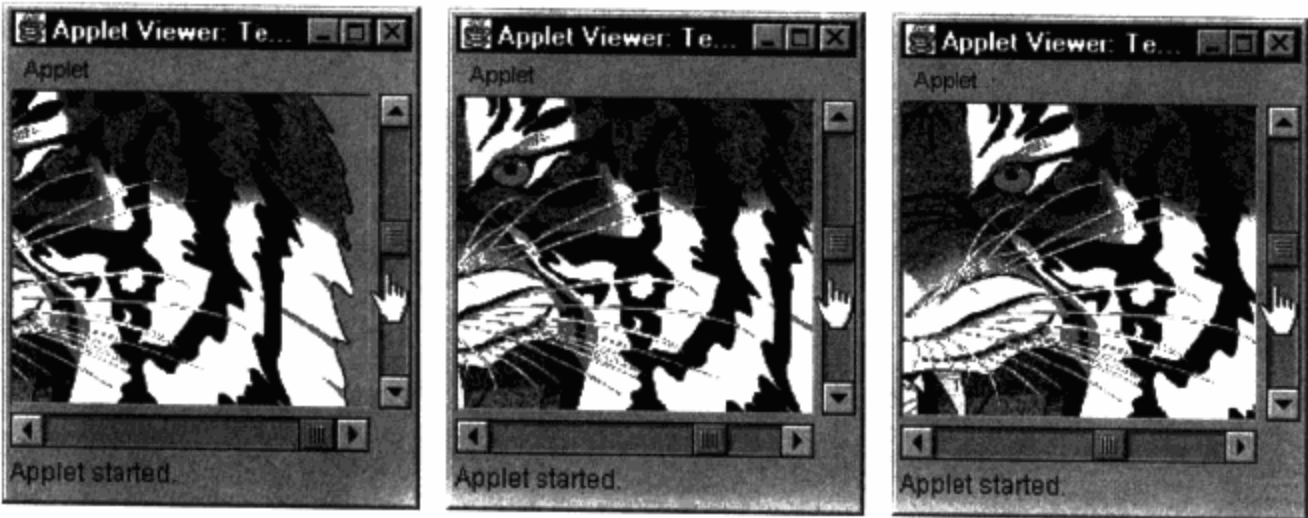


图 4-5 定制组件中的自动滚动

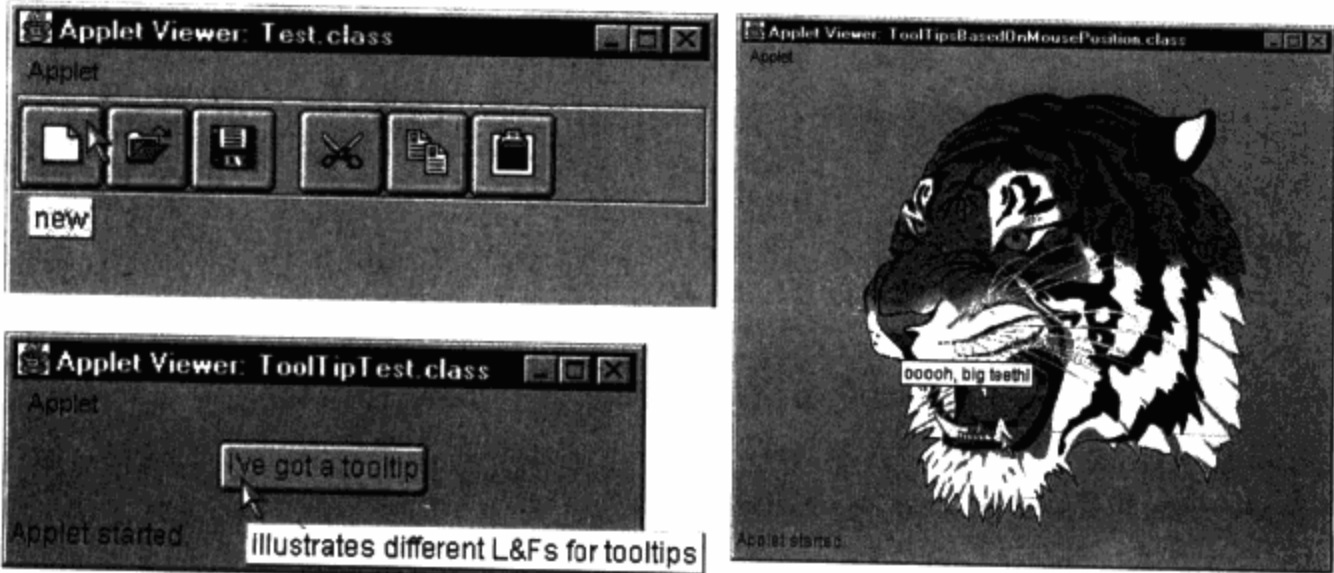


图 4-6 工具提示

4.1.6 工具提示

工具提示是当光标停在带工具提示的 JComponent 扩展上时,在一个小窗口中显示的字符串。工具提示大多用在工具条按钮上,用于描述指定按钮的功能。图 4-6 示出了在不同的情况下使用工具提示的情况。

工具提示相当灵活,例如,可根据停留在组件上的光标位置来改变工具提示的文本内容。图像映像是需要这种特性的主要情况,图像映像弹出对当前光标下的内容的简单说明。例如,可用工具提示来实现 VH1 的弹出视频。

可通过 ToolTipManager 类来设置从光标开始停留在组件上到显示工具提示之间的时间间隔。

4.1.7 键击处理和客户属性

JComponent 类提供对嵌套键击处理的支持。不需要开发人员过滤所有的键击事件并对感兴趣的键击事件作出反应,当在某些条件下按下或释放一个键击时,可以指示 JComponent 把动作事件发送给一个特定的目标。

每个 JComponent 对象都维护一个称作客户属性的词典。词典维护一组“关键字/值”对,“关键字/值”对可以是任何类型的对象。客户属性可以把任何 JComponent 与一个不扩展 JComponent 类的对象联系起来。

4.2 JComponent 类结构

图 4-7 示出了 JComponent 类及其扩展的继承关系图。

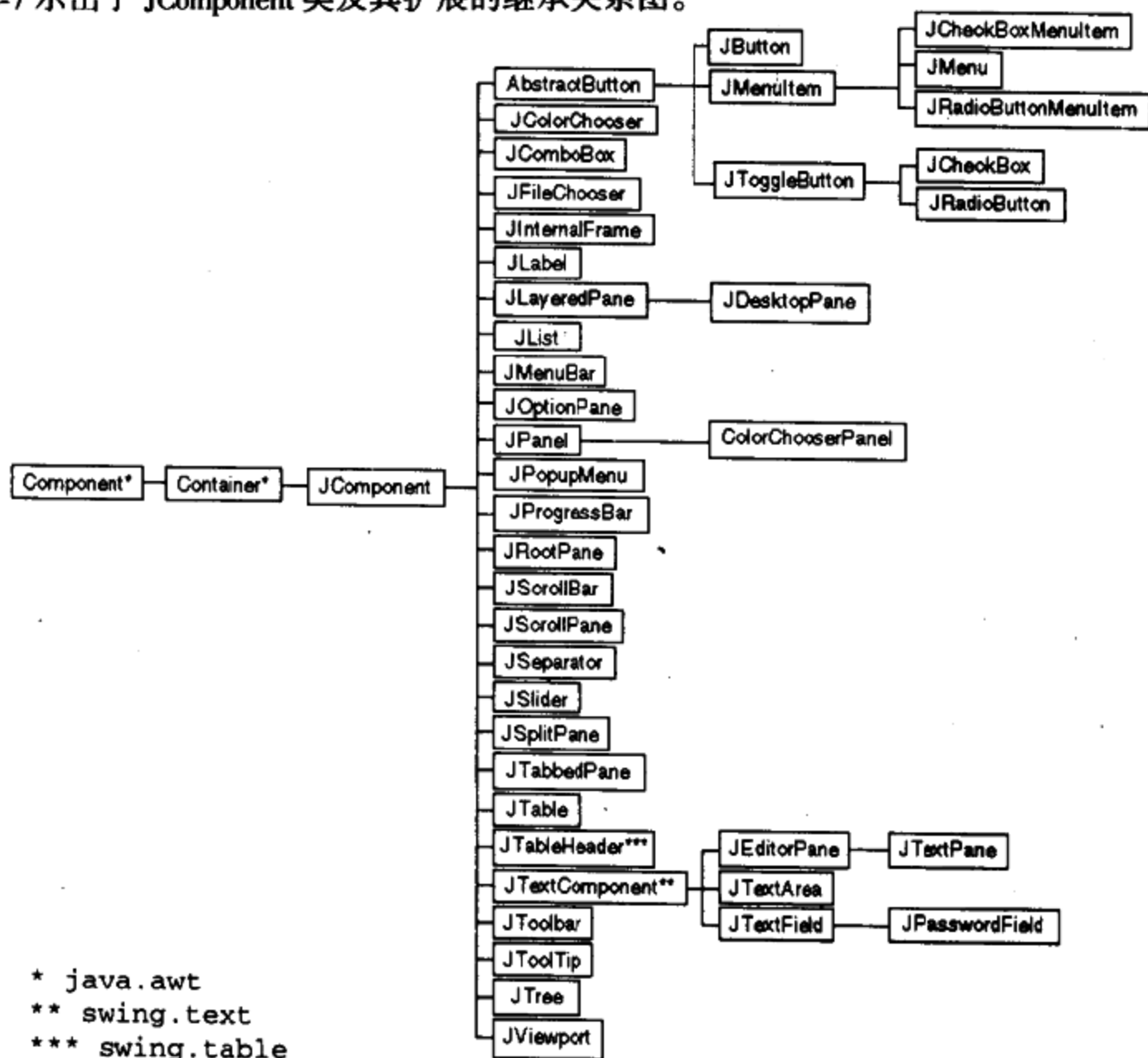


图 4-7 JComponent 的继承结构

Swing 的所有 J 类（名字以 J 开头的轻量组件）最终都是从 `swing.JComponent` 中派生出来的。除了图中注明的外，图 4-7 所示的所有类都在 `swing` 包中。

关于 `JComponent` 的层次结构有许多地方要说明一下。

首先，图 4-7 中没有明确指出：29 个 `JComponent` 扩展中的每一个都从其 `Component/Container/JComponent` 超类链中继承了 277 个 `public` 方法。这样，所有的 `JComponent` 扩展从一开始就具有了相当多的功能。

其次，继承在这个 `JComponent` 层次结构中的使用是很节省的。只有 `JTextComponent` 和 `AbstractButton` 类的后面才有大量的分支。大多数 `JComponent` 扩展都是具体组件，使用它们不需要进一步扩展。在 `JComponent` 的层次结构中，组合比继承更受欢迎，这产生了一个更灵活的结构。

4.2.1 Swing 组件是 AWT 容器

因为 `JComponent` 扩展 `java.awt.Container`，所以每个 `JComponent` 扩展都可以包含 AWT 和 Swing 组件。图 4-8 显示了一个小应用程序，该小应用程序包含一个 Swing 按钮，而该按钮又包含一个 AWT 按钮和另一个 Swing 按钮。例 4-1 列出了这个小应用程序的代码。

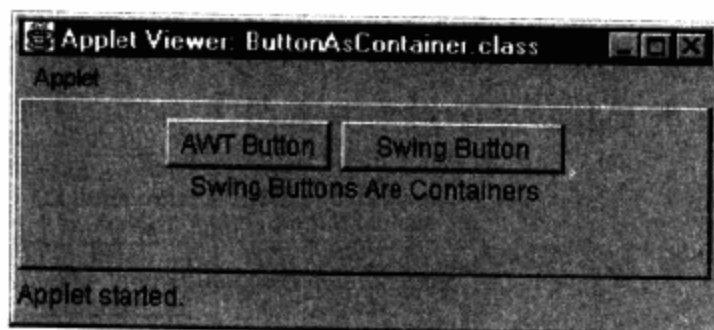


图 4-8 所有的 `JComponent` 都是 AWT 容器

声明：不要把下面的代码用于工作中。

例 4-1 作为容器使用的 Swing 按钮

```
import javax.swing.*;
import java.awt.*;

public class ButtonAsContainer extends JApplet {
    public void init() {
        JButton b = new JButton("Swing Buttons Are Containers");
        b.setLayout(new FlowLayout());
        b.add(new Button("AWT Button"));
        b.add(new JButton("Swing Button"));
        getContentPane().add(b);
    }
}
```

虽然在一个按钮中嵌入另一个按钮的设计可能会遇到问题，但是，`JComponent` 扩展 `java.awt.Container` 的事实使复合组件的实现容易多了。

4.2.2 最小尺寸、最大尺寸和首选尺寸

和 AWT 组件一样，Swing 组件也有最小尺寸、最大尺寸和首选尺寸，它们通常由布局管理

器用来设置组件的大小。

我们要对上面这句话进行解释,一个组件的最小尺寸、最大尺寸和首选尺寸并不决定其实际尺寸。通常,布局管理器设置组件的大小,且布局管理器有权决定是否使用最小尺寸、最大尺寸和首选尺寸。事实上,最小尺寸、最大尺寸和首选尺寸是一个尺寸请求,在确定组件尺寸时,该请求可能被考虑也可能不被考虑。

与 AWT 组件必须扩展才能修改最小尺寸、最大尺寸和首选尺寸不同,Swing 组件带有设置组件尺寸的方法,如表 4-1 所示。

表 4-1 用于处理组件大小的 JComponent 方法

方法名	实现
void setMaximumSize(Dimension)	设置组件的最大尺寸
void setMinimumSize(Dimension)	设置组件的最小尺寸
void setPreferredSize(Dimension)	设置组件的首选尺寸
Dimension getMaximumSize ()	返回组件的最大尺寸
Dimension getMinimumSize ()	返回组件的最小尺寸
Dimension getPreferredSize ()	返回组件的首选尺寸

表 4-1 中列出的多个 get 方法都使用下面的运算规则来计算各自的大小:

```
public Dimension getPreferredSize () {
    if (size has been explicitly set via call to setPreferredSize ())
        return size explicitly set via call to setPreferredSize ()

    if (UI delegate is non-null)
        return size calculated by UI delegate

    if (UI delegate is null && size has not been explicitly set)
        return superclass implementation of getPreferredSize ()
}
```

如果显式地设置组件的尺寸,则该尺寸值从 get 方法中返回。

如果没有显式地设置组件的尺寸而且组件有一个 UI 代表,则调用这个 UI 代表来计算组件的尺寸。

如果没有显式地设置组件的尺寸且组件没有 UI 代表,则调用超类来计算组件的尺寸。因为 JComponent 扩展是 AWT 容器,所以 java.awt.Container 委托组件的布局管理器来计算组件的首选尺寸。

图 4-9 示出了一个小应用程序,它允许在弹出列表时设置 JList 实例的首选尺寸。如果首选尺寸被设置为 null,则该列表的 UI 代表计算该列表的首选尺寸。如果首选尺寸被设置为 (100, 100),则该列表的尺寸从 getPreferredSize 方法中返回。

这个小应用程序内容窗格的布局管理器设置为 FlowLayout 的一个实例,这个实例根据组件的首选尺寸来设置组件的大小。

如果列表的首选尺寸是 null,则用该列表的 UI 代表来计算列表的大小,如图 4-9 上图所示。很明显,列表的 UI 代表掌握着列表的大小,列表的大小设置为刚好放下列表中的项。

如果列表的首选尺寸被显式地设置为 100 × 100 像素,则 UI 代表超出了这幅图,列表被设置为 (100, 100) 个像素。

例 4-2 列出了图 4-9 所示的小应用程序的代码。

例 4-2 为 JComponent 显式地设置首选大小

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    JComboBox sizeCombo = new JComboBox (new Object [] {
```

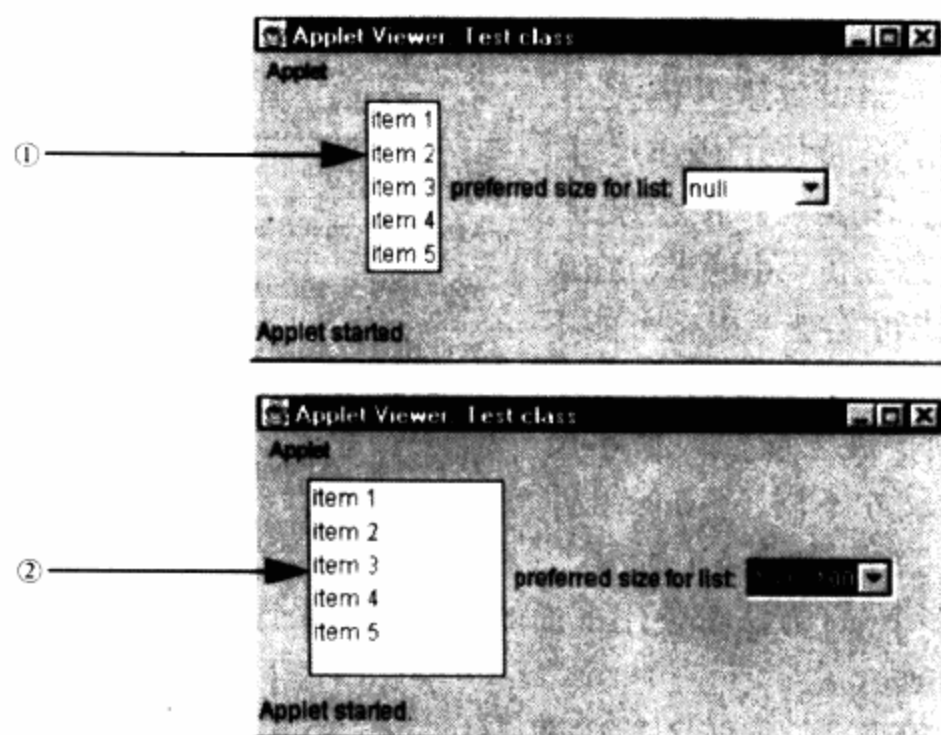


图 4-9 为 JComponent 扩展设置首选尺寸

```

        "null",
        "100 x 100"
    );
    JList list = new JList (new Object [] {
        "item 1",
        "item 2",
        "item 3",
        "item 4",
        "item 5",
    });
    public void init () {
        final Container contentPane = getContentPane ();
        list.setBorder (
            BorderFactory.createLineBorder (Color.black));
        sizeCombo.setSelectedIndex (0);
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (list);
        contentPane.add (new JLabel ("preferred size for list:"));
        contentPane.add (sizeCombo);
        sizeCombo.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int index = sizeCombo.getSelectedIndex ();
                if (index == 0)
                    list.setPreferredSize (null);
                else
                    list.setPreferredSize (
                        new Dimension (100, 100));
                // force the content pane to update
                list.revalidate ();
            }
        });
    }
}

```

当在组合框中选取一个选项时，列表的首选尺寸因此被设置，列表重新生效，结果，重新调整列表大小并重新绘制列表。有关 `revalidate` 方法的更多信息，请参见 4.3.5 节“`validate`、`invalidate` 和 `revalidate` 方法”。

Swing 提示

Swing 的 JComponent 使尺寸属性更加好用

`java.awt.Component` 类为最小尺寸、最大尺寸和首选尺寸提供了 `get` 方法，但没有相应的 `set` 方法。例如，有 `getPreferredSize` 方法，但没有相应的 `setPreferredSize` 方法。这意味着最小/最大/首选尺寸只能在每个类的基础上设置，即需要重载扩展类中的 `get` 方法。例如，为 AWT 图片设置首选大小需要使用下面的代码：

```
// SomeCanvas has preferred size of 100 x 100 pixels...
SomeCanvas canvas = new SomeCanvasExtension();
...
class SomeCanvasExtension extends Canvas {
    public Dimension getPreferredSize() {
        return new Dimension(100, 100);
    }
}
```

`JComponent` 类提供了 `set` 方法，以便最小/最大/首选尺寸可在每个实例的基础上设置，而不是在每个类的基础上设置。例如，`JPanel` 用下面的代码设置首选大小：

```
JPanel panel = new JPanel();
...
panel.setPreferredSize(new Dimension(100, 100));
```

感谢 `JComponent`，使我们不再需要使用继承来修改组件的最小尺寸、最大尺寸和首选尺寸。

4.3 绘制 JComponent 组件

Swing 轻量组件的绘制是组件和组件 UI 代表合作的结果。通常，开发人员不需要涉及绘制

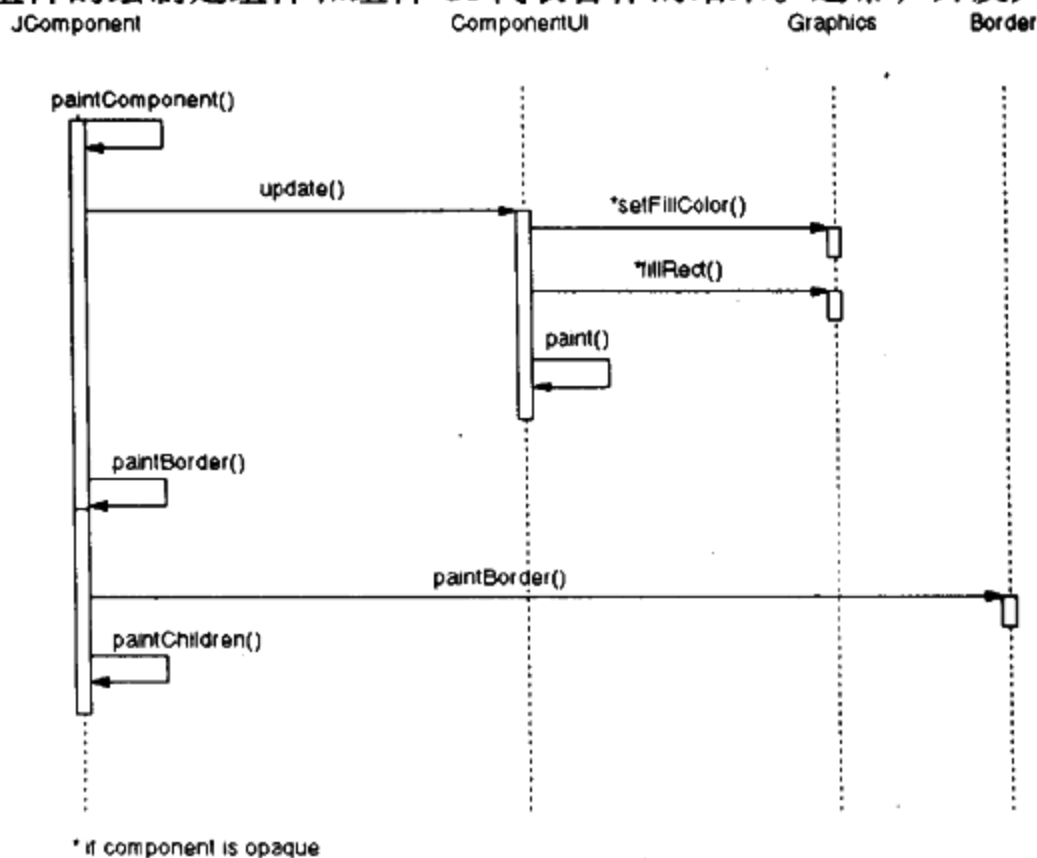


图 4-10 JComponent.paint() 方法的顺序图

过程，轻量组件被简单地实例化并添加到一个容器中。在恰当的时候，该容器会布局它所含的组件并绘制这些组件。然而，有时理解轻量组件的绘制过程是很重要的。例如，更新一个组件的可视表现或扩展 JPanel 类以绘制定制的图形都要求对绘制过程有一定的理解。

用 JComponent.paint 方法来绘制轻量 Swing 组件，JComponent.paint 方法以一个要绘制的图形作为参数。JComponent.paint 要考虑组件是否可以双缓存，是否可以有边框，是否可以包含其他组件。图 4-10 给出了由 JComponent.paint 触发方法的调用顺序。

JComponent.paint 先绘制组件，然后绘制组件的边框，再绘制组件的子组件。调用的顺序确保组件、边框和子组件都是可视的。

如果组件有一个 UI 代表，则 JComponent.paintComponent 调用该代表的 Update 方法，该方法为不透明组件擦除背景，然后绘制组件。

UI 代表和组件的边框都以一个用来绘制组件的实际图形的拷贝为参数。因此，paintBorder 和 paintChildren 方法可以改变传送给它们的图形，而不影响接下来的图形操作。有关拷贝 Graphics 和设置它们的参数的更多信息，请参见《Java2 图形设计，卷 I: AWT》。

4.3.1 Swing 组件中的定制绘制

如果读者有使用 AWT 的经验，则应该知道为了绘制定制组件必须重载 java.awt.Component.paint。

Swing 轻量组件的定制绘制要稍微复杂些。因为 Swing 组件是容器，而且该容器可能有子组件和边框，更别提 UI 代表了。为了说明 AWT 组件中的定制绘制与 Swing 组件中的定制绘制的差别，本节介绍一个简单的 ImageCanvas 类的实现过程，首先用 AWT 的方法来实现，然后用 Swing 的方法来实现。

注意 下节中实现的 ImageCanvas 类只是为了说明问题。与 AWT 不同，Swing 提供了一个可显示图像的组件 (JLabel 类)。

4.3.2 在 AWT 组件中重载绘制方法

通常，通过扩展 java.awt.Canvas 或 java.awt.Panel 并重载 paint() 来实现定制 AWT 组件。例如，图 4-11 显示了一个小应用程序，它包含一个 ImageCanvas 的实例，该实例是 java.awt.Canvas 的一个扩展。

ImageCanvas 类有三种功能：装载要显示的图像；重载 paint 方法以绘制该图像；重载 getPreferredSize 方法，以便画布的首选大小与该图像的大小相同。例 4-3 列出了图 4-1 所示的小应用程序和 ImageCanvas 类。

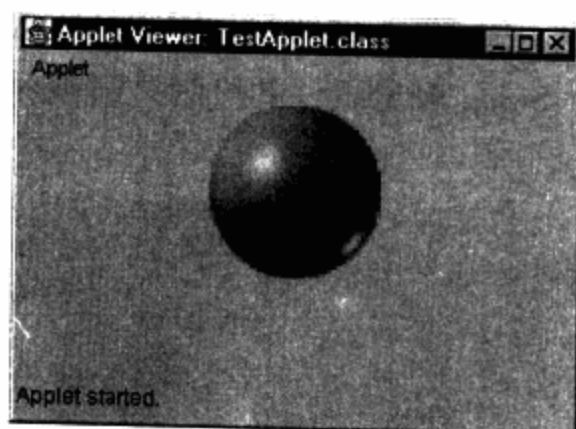


图 4-11 扩展 Java.awt.Canvas 的 ImageCanvas 类

例 4-3 为定制 AWT 组件而重载 paint()

```
import java.applet.Applet;
import java.awt.*;

public class TestApplet extends Applet {
    public void init() {
        ImageCanvas imageCanvas = new ImageCanvas ("sphere.gif");
        add (imageCanvas);
    }
}
```

```

|
class ImageCanvas extends Canvas |
    Image image;

    public ImageCanvas (String imageName) |
        image = Toolkit.getDefaultToolkit().getImage (imageName);
        MediaTracker mt = new MediaTracker (this);
        try |
            mt.addImage (image, 0);
            mt.waitForID (0);
        |
        catch (InterruptedException ex) |
            ex.printStackTrace ();
        |
    |

    public void paint (Graphics g) |
        g.drawImage (image, 0, 0, null);
    |

    public Dimension getPreferredSize () |
        return new Dimension (image.getWidth (null),
                                image.getHeight (null));
    |
|

```

ImageCanvas 的构造方法以一个字符串为参数，这个字符串是图像文件的文件名。使用一个 MediaTracker 实例来确保在构造方法返回前，图像已经完全装载好了。

重载 paint 方法以便只绘制画布左上角图像。Graphics.drawImage ()、Image.getWidth () 和 Image.getHeight () 都以一个图像观察器的 null 引用为参数，因为在调用 paint () 前，已完全装载了这个图像。

ImageCanvas 重载 getPreferredSize () 方法以便返回图像的大小。AWT 小应用程序的缺省布局管理器是 FlowLayout，FlowLayout 根据组件首选大小来设置组件大小。因此，图像画布的大小设置成要显示的图像大小。

4.3.3 在 Swing 组件中重载绘制方法

图 4-12 显示了一个与图 4-11 相似的小应用程序。

图 4-12 所示的小应用程序创建一个 ImageCanvas 实例，该实例是从 Swing.JPanel 类中派生出来的。该小应用程序创建一个带标题边框（在 Swing 的边框制造器的帮助下），并把这个边框作为图像画布的边框。然后，这个图像画布添加到小应用程序的内容窗格中。例 4-4 列出了该小应用程序的代码。

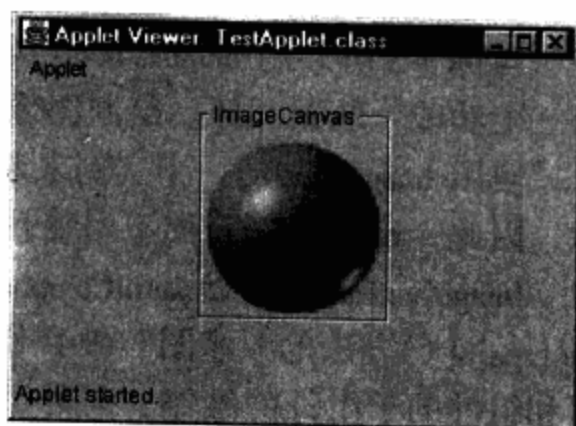


图 4-12 扩展 JPanel 的 ImageCanvas 类

例 4-4 带有从 JComponent 派生的图像画布的小应用程序

```

public class TestApplet extends JApplet |
    public void init () |
        container contentPane = getContentPane ();
        ImageCanvas imagePanel = new ImageCanvas ("sphere.gif");

```



```

imagePanel.setBorder (
    BorderFactory.createTitledBorder ("ImageCanvas"));

contentPane.setLayout (new FlowLayout ());
contentPane.add (imagePanel);
}
}

```

这个小应用程序还把它的内容窗格的布局管理器设置为一个 `FlowLayout` 实例，因此，这个图像画布的大小设置为其首选尺寸。与 `java.awt.Applet` 不同，缺省时，`JApplet` 使用一个 `BorderLayout` 实例。

例 4-5 列出了 `ImageCanvas` 类。

例 4-5 为定制 Swing 组件而重载 `paintComponent()`

```

class ImageCanvas extends JPanel {
    ImageIcon icon;

    public ImageCanvas (String imageName) {
        icon = new ImageIcon (imageName);
    }

    public void paintComponent (Graphics g) {
        super.paintComponent (g);

        Insets insets = getInsets ();
        icon.paintIcon (this, g, insets.left, insets.top);
    }

    public Dimension getPreferredSize () {
        Insets insets = getInsets ();
        return new Dimension (
            icon.getIconWidth () + insets.left + insets.right,
            icon.getIconHeight () + insets.top + insets.bottom);
    }
}

```

与例 4-3 所列的 `ImageCanvas` 类一样，例 4-4 列出的从 `swing.JPanel` 派生的 `ImageCanvas` 类也有三个功能：装载图像、绘制图像和指定画布的首选大小。

使用 `ImageIcon` 的一个实例来装载图像。当指定了图像时，图像图标将完全装载它们的图像，因此，`ImageCanvas` 类不用装载图像。有关图标的更多信息，请参见 5.2 节“图标”。

`ImageCanvas` 类重载 `paintComponent()` 方法，而不是重载 `paint()` 方法。从图 4-10 “`JComponent.paint()` 方法的顺序图”中我们可以看到，`JComponent.paint()` 能很好地完成组件、组件边框和组件的子组件的绘制。另外，对双缓存组件，`JComponent.paint()` 负责把组件绘制到屏外缓存中，然后把屏外缓存拷贝到组件的屏上代表中。正因为如此，所以我们不建议为 Swing 组件重载 `paint()`。如果需要重新定义如何绘制组件，那么就重载 `paintComponent()`。

`ImageCanvas.paintComponent()` 方法调用 `super.paintComponent()` 方法。对不透明组件而言，这是必须的，因为背景由组件的 UI 代表或超类的 `paintComponent` 方法擦除[○]。另外，如果该组件有一个 UI 代表，那么，如果不调用 `super.paintComponent()`，UI 代表将没有机会进行绘制组

○ 不是所有的 Swing 组件都有一个 UI 代表。如果没有 UI 代表，则组件自己来处理擦除背景的工作。

件时它需要完成的那部分工作。参见图 4-10 “JComponent.paint()方法的顺序图”。

与例 4-3 中的 ImageCanvas 版本不同, 本例中的 ImageCanvas.paintComponent 不把图像绘制在面板的左上角上, 而是把图标绘制在 (insets.left, insets.top) 位置上, 这里的 insets 指面板的边衬。所有的 Swing 轻量组件实际上都是 AWT 容器, 所以应避免把图像绘制到组件的边衬中。例如, 在例 4-5 中, 如果在位置(0,0)上绘制图标, 那么边框就会绘制到图像的外边缘上。

当计算面板的首选大小时, 重载 getPreferredSize 方法也要考虑到面板边衬的问题。例 4-3 所列的 AWT ImageCanvas 版本的首选大小就是图像的大小。而 JPanel 的扩展则必须确保其首选大小把由边衬定义的区域也考虑进去了。

Swing 提示

作为 JLabel 显示的图像

图像不是组件, 因此需要前面介绍的 ImageCanvas 类。这是 AWT 做事的方法; 而 Swing 使用 JLabel 类的一个实例来显示图像。与所有的 Swing 组件一样, JLabel 也是一个 AWT 组件, 因此, 很容易嵌入到 Swing 和 AWT 容器中。

4.3.4 paint、repaint 和 update 方法

调用 repaint() 后, 就会调用 AWT 组件和 Swing 组件的 update 方法。对 AWT 组件, update() 擦除该组件的背景, 接着调用 paint()。有时, 擦除背景是人们所不希望的。例如, 在一个图形条中, 不断用获得的新数据来更新其显示。反复擦除和重新绘制该图像会导致可察觉的闪烁。在这种情况下, 一般是用 AWT 组件的扩展来重载 update() 以便直接调用 paint() 方法, 因此, 避免了擦除组件背景和与之有关的闪烁。

对 Swing 组件而言, 为了消除闪烁, 不需要重载 update() 来直接调用 paint()。因为, JComponent 类重载了 update(), 直接调用 paint()。然而, 这不是说 Swing 组件在重新绘制组件时不擦除组件背景。当重新绘制 Swing 组件时, 擦除这个组件背景的任务就交给这个组件的 UI 代表来完成 (如果这个组件有一个 UI 代表的话)。

幸运的是, 不需要子类化一个组件的 UI 代表来消除闪烁, 因为, 缺省时, Swing 组件是双缓存的, 这意味着擦除组件然后重新绘制组件的工作都是在屏外缓存中进行的。当组件在其屏外缓存中更新后, 再把更新后的组件拷贝到屏幕上。双缓存消除了闪烁, 因此, 不需要为 Swing 组件重载 update() 方法。如果 Swing 组件闪烁, 则解决方法是确保该组件是双缓存而不是重载 update()。

4.3.5 validate、invalidate 和 revalidate 方法

Swing 和 AWT 组件通常由它们容器的布局管理器来定位和给出大小 (布局)。在任何给定时刻, 一个组件不是有效的就是无效的; 无效的组件需要被布局, 而有效的组件则不需要被布局。例如, 如果在一个按钮构造后要设置这个按钮的文本, 则很可能该按钮需要重新安排大小以容纳下这个新文本。因此, 设置按钮的文本导致该按钮是无效的 (通过为该按钮设置 invalidate(), 指示该按钮需要被布局)。

Swing 和 AWT 的 containers 都提供了一个 validate 方法, 该方法可为容器中的所有组件布局。例如, 有一个容器, 它含有一个按钮, 且这个按钮的文本已经改变 (因此是无效的), 如果为该容器调用 validate(), 则该容器将布局这个按钮及该容器中的所有其他组件。

对 `invalidate()` 和 `validate()` 的调用在容器层次结构中分别向上和向下传播, 因此, 使一个组件无效会使这个组件的容器层次结构中的所有容器无效。

`JComponent` 类提供了一个 `revalidate` 方法, 这个方法使组件无效并为组件层次结构中的第一个容器在事件派发线程上安排一个 `validate()` 的调用, 如果组件的 `isValidateRoot` 属性被设置为 `true` 的话。因为包含在 `JApplet` 或 `JFrame` 实例中的所有 Swing 组件都在根面板中, 所以, 为 Swing 小应用程序或应用程序中的一个 Swing 组件调用 `revalidate()` 将使该组件被布局。在组件布局后, 已改变的组件区域将重新绘制。

关于 `JComponent.revalidate()`, 有几点要注意。

第一, 同调用 `repaint()` 一样, 对 `revalidate()` 的调用是合并的。因此, 如果为某个容器中的多个组件连续调用 `revalidate()`, 则这些 `revalidate()` 调用将合并, 即只调用一次 `revalidate()`。

第二, 任何为改变位置和大小而改变的 Swing 组件应该通过组件本身来重新生效, 布局和重新绘制组件。不幸的是, 对 Swing 1.1 FCS/1.2 JDK, 重新生效操作并不总是执行。如果在组件的位置和大小改变后, 组件没有被布局 and 重新绘制, 则这是 Swing 的错误造成的。

第三, 虽然对 `JComponent.revalidate()` 的调用应该使已经变化了的组件域重新绘制, 但是重新绘制不总是发生。因此, 有时需要在调用 `revalidate()` 后, 再调用 `repaint()`。

Swing 提示

在 Swing 组件中定制绘制

在 Swing 组件中定制绘制比在 AWT 组件中定制绘制要复杂多了, 因为 Swing 组件是可以带边框的轻量 AWT 容器。下面是有关绘制 Swing 组件要记住的一些基本原则:

- 1) 如果需要控制组件本身的绘制过程, 但又想用缺省的操作来绘制该组件的边框和子组件, 则重载 `JComponent.paintComponent()`。
- 2) 如果需要完全控制组件、其边框和其子组件的绘制过程, 请重载 `JComponent.paint()`。
- 3) 在重载的 `paintComponent` 方法中调用 `super.paintComponent()` 以确保 UI 代表获得擦除不透明组件的背景和绘制的机会。
- 4) 重载 `paintComponent` 方法时, 不要绘制到组件的边衬中。
- 5) 在计算最小、最大或首选尺寸时, 要考虑组件的边衬的大小。
- 6) 如果 Swing 在更新时闪烁, 则应确保该组件是双缓存的, 而不是重载它的更新方法。

4.3.6 不透明组件与透明组件的比较

Swing 轻量组件可以是不透明的或部分透明的。

例如, 图 4-13 所示的小应用程序包含 `ColoredPanel` 类的两个实例。左边的面板是不透明的, 而右边的面板是部分透明的。

图 4-13 所示的小应用程序实例化一个 `RainPanel` 实例, 该实例含有 `ColoredPanel` 类的两个实例。`RainPanel` 在其背景上平铺一个图像, `ColoredPanel` 绘制一个黑色的边框和一个在组件中居中的、填充了的矩形。`RainPanel` 和 `ColoredPanel` 都扩展 `JPanel`。例 4-6 列出了该小应用程序的代码。

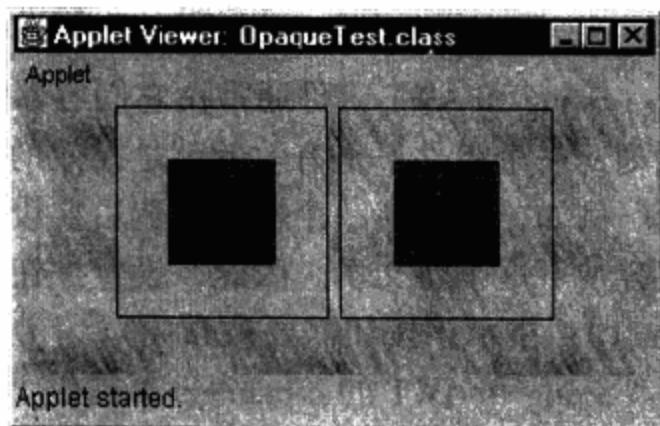


图 4-13 一个不透明的组件和一个部分透明的组件

例 4-6 不透明测试小应用程序

```

public class OpaqueTest extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        RainPanel rainPanel = new RainPanel ();

        ColoredPanel opaque = new ColoredPanel (),
            transparent = new ColoredPanel ();

        // JComponents are opaque by default, so the opaque
        // property only needs to be set for transparent
        transparent.setOpaque (false);

        rainPanel.add (opaque);
        rainPanel.add (transparent);

        contentPane.add (rainPanel, BorderLayout.CENTER);
    }
}

```

缺省时, Swing 组件是不透明的, 所以, 不需要为 opaque 面板设置不透明属性。该小应用程序为 transparent 面板调用 `setOpaque (false)`, 该面板允许下雨的面板的背景显示出来。如图4-10 “JComponent.paint ()方法的顺序图” 中所示, 不透明组件的背景是该组件的背景颜色, 而透明组件有透明的背景。

例 4-7 列出了图 4-13 所示的小应用程序的完整代码。

例 4-7 不透明 Swing 组件

```

import javax.swing.*;
import java.awt.*;

public class OpaqueTest extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        RainPanel rainPanel = new RainPanel ();

        ColoredPanel opaque = new ColoredPanel (),
            transparent = new ColoredPanel ();

        // JComponents are opaque by default, so the opaque
        // property only needs to be set for transparent
        transparent.setOpaque (false);

        rainPanel.add (opaque);
        rainPanel.add (transparent);

        contentPane.add (rainPanel, BorderLayout.CENTER);
    }
}

class RainPanel extends JPanel {
    ImageIcon rain = new ImageIcon ("rain.gif");
    private int rainw = rain.getIconWidth ();
    private int rainh = rain.getIconHeight ();

    public void paintComponent (Graphics g) {
        Dimension size = getSize ();

        for (int row = 0; row < size.height; row += rainh)
            for (int col = 0; col < size.width; col += rainw)

```

```

        rain.paintIcon (this, g, col, row);
    }

    class ColoredPanel extends JPanel {
        public void paintComponent (Graphics g) {
            super.paintComponent (g);

            Dimension size = getSize ();

            g.setColor (Color.black);
            g.drawRect (0, 0, size.width-1, size.height-1);

            g.setColor (Color.red);
            g.fillRect (size.width/2-25, size.height/2-25, 50, 50);

        }

        public Dimension getPreferredSize () {
            return new Dimension (100, 100);
        }
    }

```

回顾一下前面 Swing 提示“在 Swing 组件中定制绘制”中提到的指导方针，其中的一条建议是：在重载的 `paintComponent` 方法中调用 `super.paintComponent()` 以擦除组件的背景。例 4-7 所列的代码含有一个遵守这个指导方针的类和一个不遵守这个指导方针的类。`ColoredPanel` 类必须确保在被绘制时，其背景已填充，以便不透明实例确实是不透明的。另一方面，`RainPanel` 用一个图标平铺其背景，因此，使人察觉不到在擦除背景。结果，`ColoredPanel.paintComponent()` 调用了 `super.paintComponent()`，但 `RainPanel` 却没有调用这个方法。

4.3.7 立即绘制 Swing 组件

为一个组件调用 `repaint()` 会产生一个将要放到 AWT 事件序列上的绘制事件。因此，如果 `repaint()` 是从一个事件处理方法中调用的，则直到该事件处理方法返回才会进行重新绘制。

`JComponent` 类提供了立即绘制组件的方法，即 `paintImmediately()` 方法，该方法适合在需要立即更新组件外观的事件处理方法中使用^①。

应该有选择地使用 `paintImmediately()` 方法，因为在大多数情况下，调用 `repaint()` 更有效，因为 `repaint()` 会使多余的绘制请求失败。

图 4-14 示出了一个含有一个按钮和一个 `ColoredPanel` 实例的小应用程序。当该按钮激活时，其动作监听器休眠五秒以模仿一些费时的动作。

在继续介绍前，必须指出绝对不提倡在事件处理方法中包含费时的行为。因为事件只在一个事件派发线程上派发，所以希望事件处理方法尽快接手处理，通常，应该在一个单独的线程上调度与 GUI 无关

的费时动作。然而，有时事件处理方法需要立即更新一个组件的外观，图 4-14 所示的小应用程序中引入的延迟是为了说明 `paintImmediately` 方法的益处。

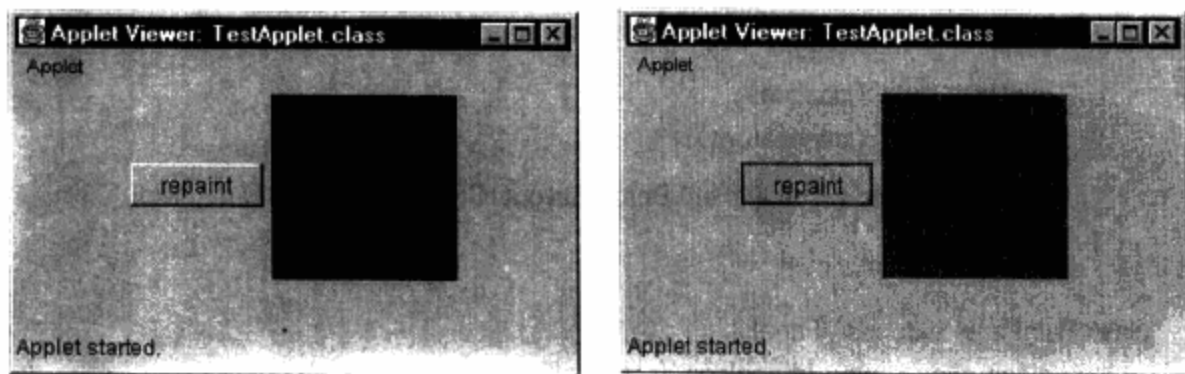


图 4-14 立即绘制 Swing 组件

① 在读者读到这里时，也许 `paintImmediately()` 方法已移植到 `java.awt.Component` 中。

该小应用程序启动时显示一个蓝颜色面板。单击“repaint”按钮导致在事件处理方法返回之前用红色来重新绘制该蓝颜色面板。例 4-8 列出了这个小应用程序的代码。

例 4-8 在事件处理方法中使用 `paintImmediately()`

```
import javax.swing.*;
import java.awt.*; ...
import java.awt.event.*;

public class TestApplet extends JApplet {
    public void init() {
        Container contentPane = getContentPane();
        final JPanel panel = new JPanel();
        JButton button = new JButton("repaint");

        panel.setBackground(Color.blue);
        panel.setPreferredSize(new Dimension(100, 100));
        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);
        contentPane.add(panel);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Color c = panel.getBackground();
                Dimension sz = panel.getSize();

                panel.setBackground(
                    c == Color.blue ? Color.red : Color.blue);

                panel.paintImmediately(
                    0, 0, sz.width, sz.height);

                // for illustrative purposes only
                try {
                    Thread.currentThread().sleep(5000);
                }
                catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
        });
    }
}
```

`paintImmediately()` 用四个 `integer` 参数来指定要重新绘制的组件的边框 (`paintImmediately()` 的一个重载版本则带一个 `Rectangle` 参数)。例 4-8 中的小应用程序指定了作为矩形绘制的组件的边框。

4.4 双缓存

轻量组件完全在 `java` 代码中绘制，与之相反，重量组件由本地对等组件绘制。因此，双缓存轻量组件是很重要的，因为只有这样才能避免在重新绘制组件时由于擦除所有或部分组件所带来的闪烁。例如，如果滚动条的滑块被拖动而滚动条不是双缓存的，则滑块将经常地被擦除和重新绘制到屏幕上，这样，就产生了无法接受的闪烁。

通过绘制到屏外缓存，再拷贝到屏幕上，双缓存消除了闪烁。《Java 2 图形设计，卷 I：

AWT》中对双缓存进行了详细介绍。

虽然所有的 Swing 轻量组件都可以对它们的显示双缓存，但是，只有两种组件缺省是双缓存的，这两个组件是：JRootPane 和 JPanel。这似乎与本节第一段的内容相左，但是，一旦清楚了 Swing 是如何实现双缓存的，就会明白不双缓存其他的 Swing 轻量组件是有道理的。下面介绍 Swing 双缓存是怎样工作的。

Swing 包括一个 RepaintManager 类，该类维护一个屏外缓存。该缓存很大，足以容纳下屏幕的大小，用它来双缓存所有的 Swing 轻量组件。如果一个 JComponent(或这个组件的容器层次结构中的一个父容器)是双缓存的，则该组件绘制到重新绘制管理器的屏外缓存中，接着拷贝到该组件的屏上代表中。如果一个组件处在一个双缓存的容器中，则该组件就自动是双缓存的，因此，对 Swing 的轻量容器来说，缺省时，只需规定 JRootPane 和 JPanel 是双缓存的即可。

有两种方法来控制组件是否是双缓存的。第一种方法是 JComponent 类提供一个 setDoubleBuffered 方法，该方法带一个 boolean 参数值，用于指示组件是否为双缓存的。第二种方法是 RepaintManager 类提供一个 setDoubleBufferingEnabled 方法，该方法也带一个 boolean 参数，该值立即允许或禁止所有的 Swing 轻量组件是双缓存的^①。

图 4-15 所示的小应用程序含有一个滑杆，它的双缓存状态由复选框控制。

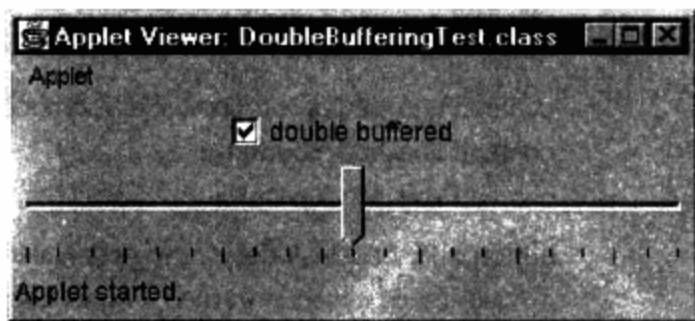


图 4-15 允许或禁止 JSlider 为双缓存

例 4-9 列出了图 4-15 所示的小应用程序的代码。

例 4-9 双缓存测试小应用程序

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class DoubleBufferingTest extends JApplet {
    public void init() {
        final JSlider slider =
            new JSlider(JSlider.HORIZONTAL, 0, 100, 50);

        final Container contentPane = getContentPane();
        JCheckBox dbcheckBox = new JCheckBox("double buffered");
        JPanel controlPanel = new JPanel();

        dbcheckBox.setSelected(true);
        controlPanel.add(dbcheckBox);

        slider.setPaintTicks(true);
        slider.setMinorTickSpacing(5);
        slider.setMajorTickSpacing(15);

        contentPane.add(controlPanel, "North");
        contentPane.add(slider, "Center");

        dbcheckBox.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent event) {
                if (event.getStateChange() == ItemEvent.SELECTED) {
                    slider.setDoubleBuffered(true);
                }
                else {

```

① 实际上，该方法影响属于相同线程组的所有组件。每个线程组有一个重新绘制管理器。

```

        slider.setDoubleBuffered (false);
    }
}
);
}
}

```

如果从 CD 中运行例 4-9 所示的小应用程序，可能会惊奇地发现虽然为滑杆设置了双缓存，但在滑杆显示的问题上却无效果。不管滑杆的双缓存是否是允许的，当其滑块移动时，滑块无闪烁地重新绘制。

虽然在复选框被选中或未选中时，滑杆的双缓存确实分别是允许的和禁止的，但都是不合理的，因为滑杆在双缓存容器中显示。我们知道，JApplet 实例包含一个 JRootPane 实例，而 JRootPane 又包含一个 JPanel 实例，JPanel 是添加滑杆的内容窗格。如前所述，JRootPanel 和 JPanel 在缺省时是双缓存的，而且因为双缓存容器中的组件无条件地都是双缓存，所以为滑杆禁用双缓存不会影响滑杆的绘制效果。

为使滑杆直接绘制到屏幕上，必须使小应用程序的根面板和内容窗格的双缓存特性都被禁用，如例 4-10 所列出的小应用程序那样。

例 4-10 为组件的容器禁用双缓存

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class DoubleBufferingTest extends JApplet {
    public void init () {
        final Container contentPane = getContentPane ();

        JSlider slider = new JSlider (JSlider.HORIZONTAL, 0, 100, 50);
        JCheckBox dbcheckBox = new JCheckBox ("double buffered");
        JPanel controlPanel = new JPanel ();

        dbcheckBox.setSelected (true);
        controlPanel.add (dbcheckBox);

        slider.setPaintTicks (true);
        slider.setMinorTickSpacing (5);
        slider.setMajorTickSpacing (15);

        contentPane.add (controlPanel, "North");
        contentPane.add (slider, "Center");

        dbcheckBox.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {
                JComponent cp = (JComponent) getContentPane ();
                JComponent rp = (JComponent) getRootPane ();

                if (event.getStateChange () == ItemEvent.SELECTED) {
                    rp.setDoubleBuffered (true);
                    cp.setDoubleBuffered (true);
                }
                else {
                    rp.setDoubleBuffered (false);
                    cp.setDoubleBuffered (false);
                }
            }
        });
    }
}

```

```

    }
    });
}

```

例 4-10 所示的小应用程序没有直接设置滑杆的双缓存状态。允许和禁止滑杆所在的容器的双缓存状态将决定滑杆是否是双缓存的。如果运行例 4-10 所示的小应用程序，将注意到当禁用双缓存且拖动滑块时，滑杆的滑块明显地闪烁。

另一个控制双缓存的方法是同时允许或禁用所有组件的双缓存特性。例 4-11 所示的小应用程序就采用这种方法，它为与这个小应用程序相关联的重新绘制管理器调用 `setDoubleBufferingEnabled`。

例 4-11 调用 `RepaintManager.setDoubleBufferingEnabled()`

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class DoubleBufferingTest extends JApplet {
    public void init() {
        Container contentPane = getContentPane();
        JCheckBox dbcheckBox = new JCheckBox("double buffered");
        JPanel controlPanel = new JPanel();
        final JSlider slider =
            new JSlider(JSlider.HORIZONTAL, 0, 100, 50);

        dbcheckBox.setSelected(true);
        controlPanel.add(dbcheckBox);

        slider.setPaintTicks(true);
        slider.setMinorTickSpacing(5);
        slider.setMajorTickSpacing(15);

        contentPane.add(controlPanel, "North");
        contentPane.add(slider, "Center");

        dbcheckBox.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent event) {
                RepaintManager rm =
                    RepaintManager.currentManager(slider);

                if (event.getStateChange() == ItemEvent.SELECTED) {
                    rm.setDoubleBufferingEnabled(true);
                }
                else {
                    rm.setDoubleBufferingEnabled(false);
                }
            }
        });
    }
}

```

前面讨论了启用和禁用单个组件的双缓存的缺点，实际中很少这样做。Swing 双缓存的实现确保了轻量组件缺省时就是双缓存的。

在定制组件中使用双缓存

幸运的是，Swing 的重新绘制管理器提供了对前述屏外缓存（用于双缓存 Swing 组件）的访问。屏外缓存有多种用途。例如，图 4-16 所示的小应用程序使用屏外缓存来实现一个简单的、不闪烁的动画。

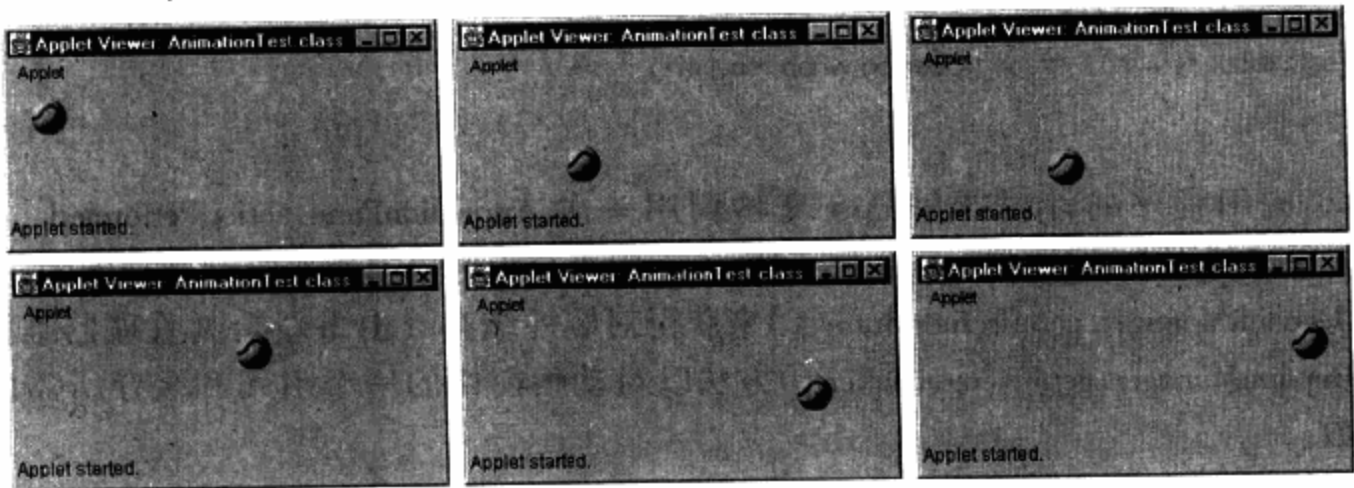


图 4-16 使用 Swing 双缓存的简单动画

该小应用程序实例化一个 AnimationPane 实例，并把该实例添加到这个小应用程序的内容窗格中。AnimationPane 类使用由 Swing 的 RepaintManager 维护的屏外缓存来画一个篮球动画图像。一个计时器也被实例化并与动画窗格相关联。有关 Swing 计时器的更多信息，请参见 6.1 节“计时器”。

例 4-12 列出了图 4-16 所示的小应用程序的代码。

例 4-12 动画测试小应用程序

```
import javax.swing.* ;
import java.awt.event.* ;
public class AnimationTest extends JApplet {
    AnimationPane p = new AnimationPane ();
    Timer timer = new Timer (5, p);

    public void init () {
        getContentPane ().add (p, "Center");
        timer.start ();

        addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent event) {
                if (timer.isRunning ()) timer.stop ();
                else timer.start ();
            }
        });
    }
}
```

所有的操作（包括执行动画）都在 AnimationPane 类中进行，AnimationPane 类扩展 JPanel 并实现 ActionListener。

AnimationPane 实例化一个接着要动画的 ImageIcon 并调用 setOpaque (false)，以便在重画时不清除它的背景。

```

public class AnimationPane extends JPanel
    implements ActionListener {
    Icon ball = new ImageIcon ("baseball.gif");
    Point ulhc = new Point (0, 0);
    Point moveVector = new Point (1, 1);
    Dimension imsz = new Dimension (ball.getIconWidth (),
                                      ball.getIconHeight ());

    public AnimationPane () {
        setOpaque (false); //don't clear bg when repainting
    }
    ...

```

这个小应用程序的计时器每隔 5 毫秒调用一次 `AnimationPane.actionPerformed`。`actionPerformed` 方法通过调用 `RepaintManager.currentManager()` 来获得对当前重新绘制管理器的一个引用。通过调用 `RepaintManager.getoffscreenBuffer()` 来获得对屏外缓存（由重新绘制管理器维护）的一个引用，`RepaintManager.getoffscreenBuffer()` 方法以对动画窗格的一个引用和缓存所需的宽度和高度为参数。

```

...
public void actionPerformed (ActionEvent e) {
    Dimension size = getSize ();
    RepaintManager rm = RepaintManager.currentManager (this);
    Image off = rm.getoffscreenBuffer (this,
                                       size.width,
                                       size.height);

    Graphics og = off.getGraphics ();
    Graphics g = getGraphics ();
    ...

```

图像图标的位置由动画窗格来维护，并且这个位置用来清除图标以前所占据的屏外缓存的矩形。

```

    If (og != null && g != null) {
        try {
            Rectangle oldr =
                new Rectangle (ulhc.x, ulhc.y, imsz.width, imsz.height);
            og.setColor (getBackground ());
            og.fillRect (ulhc.x, ulhc.y,
                        imsz.width, imsz.height);
            ...

```

然后，进行边界冲突检查。如果该图标将要遇到动画窗格的四个边之一，那么，就修改与图标相关联的移动矢量以便该图标离开动画窗格的边缘。接着，通过把移动矢量添加到当前的左上角上来计算该图标的新位置。然后，把该图标绘制到屏外缓存的新位置上。

```

    ...
    checkForEdgeCollision (size);
    ulhc.x += moveVector.x;
    ulhc.y += moveVector.y;
    ball.paintIcon (this, og, ulhc.x, ulhc.y);
    ...

```

到目前为止，所有的绘制工作都在屏外缓存中进行的。最后，把图标以前所占据的矩形和当前所占据的矩形的并集从屏外缓存拷贝到与此动画窗格相关联的屏上缓存中。

```

    ...

```

```

        Rectangle union,
            Newr = New Rectangle (ulhc.x, ulhc.y,
                                   imsz.width, imsz.height);

        Union = oldr.union (newr);
        g.setClip (union.x, union.y,
                   union.width, union.height);
        g.drawImage (off, 0, 0, this);
    }
    finally {
        og.dispose();
        g.dispose();
    }
}

// end of ActionPerformed method

```

当把相应的区域拷贝到屏上缓存中之后，将处理描述屏外缓存和屏上缓存的 Graphics，通过调用 `getGraphics()` 来获得对 Graphics 的一个引用，这在任何时候都是必要的。

例 4-13 列出了 `AnimationPane` 类的完整代码。

例 4-13 `AnimationPane` 类清单

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Animationpane extends JPanel
    implements ActionListener {
    Icon ball = new ImageIcon ("baseball.gif");
    Point ulhc = new Point (0, 0);
    point moveVector = new point (1, 1)
    Dimension imsa = new Dimension (ball.getIconWidth(),
                                      ball.getIconHeight());

    public Animationpane () {
        setopaque (false); // don't clear bg when repainting
    }

    public void actionPerformed (ActionEvent e) {
        Dimension size = getSize();

        RepaintManager rm = RepaintManager.currentManager (this);
        Image off = rm.getOffscreenBuffer (this,
                                            size.width,
                                            size.height);

        Graphics og = off.getGraphics();
        Graphics g = getGraphics();

        if (og != null) {
            try {
                Rectangle oldr =
                    new Rectangle (ulhc.x, ulhc.y,
                                   imsz.width, imsz.height);
                og.setColor (getBackground());
                og.fillRect (ulhc.x, ulhc.y,
                            imsz.width, imsz.height);
                checkForEdgeCollision (size);
            }

```



```

        ulhc.x += moveVector.x;
        ulhc.y += moveVector.y;
        ball.paintIcon (this, og, ulhc.x, ulhc.y);

        Rectangle union;
        newr = new Rectangle ( ulhc.x, ulhc.y,
                                imsz.width, imsz.height);

        union = oldr.union (newr);
        og.setColor (getBackground());
        g.setClip (union.x, union.y,
                    union.width, union.height);
        g.drawImage (off, 0, 0, this);
    }
    finally {
        og.dispose();
        g.dispose();
    }
}

private void checkForEdgeCollision (Dimension size) {
    if (ulhc.x + imsz.width + moveVector.x > size.width ||
        ulhc.x + moveVector.x < 0)
        moveVector.x -= moveVector.x * 2;
    if (ulhc.y + imsz.height + moveVector.y > size.height ||
        ulhc.y + moveVector.y < 0)
        moveVector.y -= moveVector.y * 2;
}

```

维护单个屏外缓存的有效机制使 Swing 能对其轻量组件双缓存。如果一个轻量组件在一个双缓存容器中或一个轻量组件显式地启用双缓存, 则这个组件首先会绘制到屏外缓存中, 然后再拷贝到其屏上代表中。Swing 还可以通过调用 `RepaintManger.getOffscreenBuffer()` 来获得它所使用的屏外缓存。这使开发人员能按自己的目的使用屏外缓存, 如图 4-16 所示。

Swing 提示

Swing 和双缓存

有关 Swing 和双缓存应当记住如下事项:

- 1) 双缓存容器中的组件自动被双缓存。
- 2) `JComponent.setDoubleBuffered (boolean)`: 为单个组件启用/禁用双缓存; 参见第一条。
- 3) `RepaintManger.setDoubleBufferingEnabled (boolean)`: 为所有的组件启用/禁用双缓存。
- 4) `RepaintManger.getOffscreenBuffer (JComponent, int width, int height)`: 返回对屏外缓存的一个引用。

4.5 调试图形

发生在 AWT 组件中的, 因此, 也发生在 Swing 组件中的所有的图形操作都是通过一个 `Graphics` 实例来完成的。`paint (Graphics)` 和 `update (Graphics)` 等需要执行图形操作的方法以一

个 Graphics 实例为参数。《Java2 图形设计，卷 I: AWJ》详细介绍了 Graphics 类。

Swing 提供了 java.awt.Graphics 类的一个扩展——swing.DebugGraphics，它减慢图形操作的速率并且在每次操作前闪烁。另外，DebugGraphics 类可以输出它所执行的图形调用的日志。操作执行的速率、每次操作的闪烁次数、闪烁的颜色和要输出的日志流都是可设置的，可通过 DebugGraphics 类中的 static public 方法来设置。

在利用调试图形方面，现在我们所知道的要比我们实际所需要的多。与大多数 Swing 特性一样，调试图形已不再复杂，例如，我们不需要知道还有一个扩展 java.awt.Graphics 的 DebugGraphics 类。要使用调试图形，只需调用 JComponent 的方法 setDebugGraphicsOptions (int)，其中的整数参数代表下面的调试选项：

- DebugGraphics.LOG_OPTION
- DebugGraphics.FLASH_OPTION
- DebugGraphics.BUFFERED_OPTION
- DebugGraphics.NONE_OPTION

把 Swing 组件的调试图形选项设置为 DebugGraphics.NONE_OPTION 以外的任意选项，都会引起该组件在绘制时使用一个 DebugGraphics 实例。

选项 BUFFERED_OPTION 用于双缓存组件，它引起弹出一个外部的窗口并显示该组件的屏外缓存。对其他选项的解释留给读者作为一个练习。

注意 Swing 1.1 FCS 没有实现 BUFFERED_OPTION。

图 4-17 所示的小应用程序控制这个小应用程序顶端被禁用滑杆的调试图形参数。这个滑杆被禁用，以强调它的功能只是图解说明调试图形。

这个小应用程序的 init 方法为重新绘制管理器调用 setDoubleBufferingEnabled (false)，使这个小应用程序中的所有组件都禁用双缓存。之所以要禁用双缓存，是因为双缓存组件是画在屏外缓存中的，我们看不到它的 FLASH_OPTION 效果。

```
public class DebugGraphicsTest extends JApplet {
    private JSlider slider = new JSlider();
    boolean logIsOn = false, flashIsOn = false;

    public void init() {
        ...
        RepaintManager rm =
            RepaintManager.currentManager (slider);
        rm.setDoubleBufferingEnabled (false);
        ...
        slider.setEnabled (false);
    }
    ...
}
```

该“控制”面板包含“repaint (重新绘制)”按钮、“Flash Time (闪烁时间)”和“Log (日志)”及“Flash (闪烁)”复选框。在“控制”面板中的每个组件都配备了一个监听器。“Flash Time”滑杆有一个变化监听器，这个监听器通过调用 staticDebugGraphics.setFlashTime 方法来设置闪烁时间。

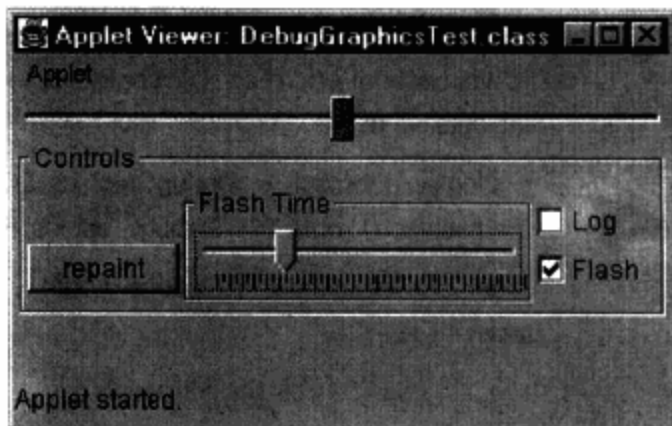


图 4-17 调试图形闪烁和记录图形操作

```
flashTimeSlider.addChangeListener (new ChangeListener () {
    public void stateChanged (ChangeEvent e) {
        DebugGraphics.setFlashTime (
            flashTimeSlider.getValue ());
    }
});
```

当被激活时,“Log”和“Flash”复选框设置由这个小应用程序维护的一些内部标志。

```
flashCheckBox.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        AbstractButton b = (AbstractButton) e.getSource ();
        if (b.isSelected ()) flashIsOn = true;
        else flashIsOn = false;
    }
});

logCheckBox.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        AbstractButton b = (AbstractButton) e.getSource ();
        if (b.isSelected ()) logIsOn = true;
        else logIsOn = false;
    }
});
```

repaint 按钮的动作监听器根据这些标志 (它们是通过复选框的选取状态来设置的) 来为禁用的滑杆设置调试选项并重新绘制这个小应用程序。

```
repaintButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        int opts = 0;
        if (logIsOn) opts |= DebugGraphics.LOG_OPTION;
        if (flashIsOn) opts |= DebugGraphics.FLASH_OPTION;
        slider.setDebugGraphicsOptions (opts);
        repaint ();
    }
});
```

例 4-14 列出了图 4-17 所示的小应用程序的完整代码。

例 4-14 为调试图形使用 Log (日志) 和 Flash (闪烁) 选项

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class DebugGraphicsTest extends JApplet {
    private JSlider slider = new JSlider ();
    boolean logIsOn = false, flashIsOn = false;

    public void init () {
        Container cp = getContentPane ();
        RepaintManager rm =
            RepaintManager.currentManager (slider);
        rm.setDoubleBufferingEnabled (false);
```

```

    cp.setLayout (new BoxLayout (cp, BoxLayout.Y_AXIS));
    cp.add (slider);
    cp.add (makeControlPanel ());
    slider.setEnabled (false);
}

private JPanel makeControlPanel () {
    JPanel controls = new JPanel ();
        checkBoxes = new JPanel ();
    JCheckBox logCheckBox = new JCheckBox ("Log"),
        flashCheckBox = new JCheckBox ("Flash");
    JButton repaintButton = new JButton ("repaint");
    final JSlider flashTimeSlider =
        new JSlider (JSlider.HORIZONTAL, 0, 250, 100);

    flashTimeSlider.setPaintTicks (true);
    flashTimeSlider.setMajorTickSpacing (10);
    flashTimeSlider.setMinorTickSpacing (5);

    controls.setLayout (new BoxLayout (controls,
        BoxLayout.X_AXIS));
    checkBoxes.setLayout (new BoxLayout (checkBoxes,
        BoxLayout.Y_AXIS));

    flashTimeSlider.setBorder (
        BorderFactory.createTitledBorder ("Flash Time"));
    controls.setBorder (
        BorderFactory.createTitledBorder ("Controls"));

    checkBoxes.add (logCheckBox);
    checkBoxes.add (flashCheckBox);

    controls.add (repaintButton);
    controls.add (flashTimeSlider);
    controls.add (checkBoxes);

    repaintButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            int opts = 0;

            if (logIsOn) opts |= DebugGraphics.LOG_OPTION;
            if (flashIsOn) opts |= DebugGraphics.FLASH_OPTION;

            slider.setDebugGraphicsOptions (opts);
            repaint ();
        }
    });

    flashTimeSlider.addChangeListener (new ChangeListener () {
        public void stateChanged (ChangeEvent e) {
            DebugGraphics.setFlashTime (
                flashTimeSlider.getValue ());
        }
    });

    flashCheckBox.addItemListener (new ItemListener () {
        public void itemStateChanged (ItemEvent e) {
            AbstractButton b = (AbstractButton) e.getSource ();

            if (b.isSelected ()) flashIsOn = true;
            else flashIsOn = false;
        }
    });
}

```

```
    |);  
    logCheckBox.addItemListener (new ItemListener () |  
        public void itemStateChanged (ItemEvent e) |  
            AbstractButton b = (AbstractButton) e.getSource ();  
            if (b.isSelected ()) logIsOn = true;  
            else      logIsOn = false;  
        |  
    |);  
    return controls;  
|  
|
```

4.6 自动滚动

JComponent 具有自动滚动的功能。当鼠标被拖出组件的边界后，自动滚动功能使组件能继续滚动。

调用 JComponent 的 setAutoScrolls (boolean) 方法可启用或禁用自动滚动。JComponent 还提供返回一个 boolean 值的 getAutoscrolls 方法，该 boolean 值指示该组件的自动滚动是否是允许的。缺省时，只有列表和表格的自动滚动是允许的。

图 4-18 示出了几个正在自动滚动的列表快照。最初，选取列表中的一项，然后鼠标向下拖动并超出了列表的边界。结果，列表继续滚动直到显示了列表的最后一项。

例 4-15 列出了该小应用程序的源代码。列表共有六项，调用 JList.setVisibleRowCount () 把可视行数设置为 3。因为 JList 缺省时是允许自动滚动的，所以，不需要为此列表调用 setAuto-scrolls (true)。该列表包裹在一个滚动窗格中，而这个滚动窗格添加到这个小应用程序中。

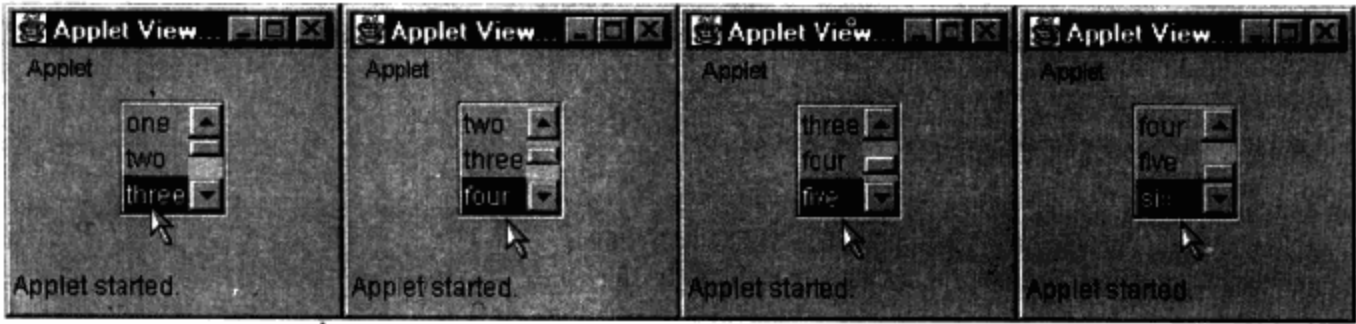


图 4-18 JList 的自动滚动
例 4-15 包裹在滚动窗格中的一个列表

```
import javax.swing. * ;  
import java.awt. * ;  
  
public class TestApplet extends JApplet |  
    public void init () |  
        String [] items = { "one", "two", "three",  
                             "four", "five", "six" };  
  
        Container contentPane = getContentPane ();  
        JList list = new JList (items);  
  
        list.setVisibleRowCount (3);  
  
        contentPane.setLayout (new FlowLayout ());
```

```
contentPane.add (new JScrollPane (list));
```

定制组件中的自动滚动

为实现定制组件的自动滚动，必须了解自动滚动是怎样工作的。当鼠标拖动超出了一个启用了自动滚动的组件时，该组件每隔 100 毫秒就发出一个鼠标拖动事件，而不管是否移动鼠标。JList 之类的组件通过自动地滚动组件的内容来处理鼠标拖动事件。一旦鼠标重新进入该组件或发生鼠标释放事件，该组件才停止发出鼠标拖动事件。

在定制组件中实现自动滚动要求该组件或组件的 UI 代表来处理发生在该组件边界外的鼠标拖动事件。例如，图 4-19 所示的小应用程序包含一个 AutoscrollViewport 实例^①，AutoscrollViewport 是 JViewport 的一个扩展。在视口中发生的鼠标拖动事件导致视口中的视图根据鼠标的移动而移动。把鼠标拖出视口将引起视图向鼠标移动的方向滚动。

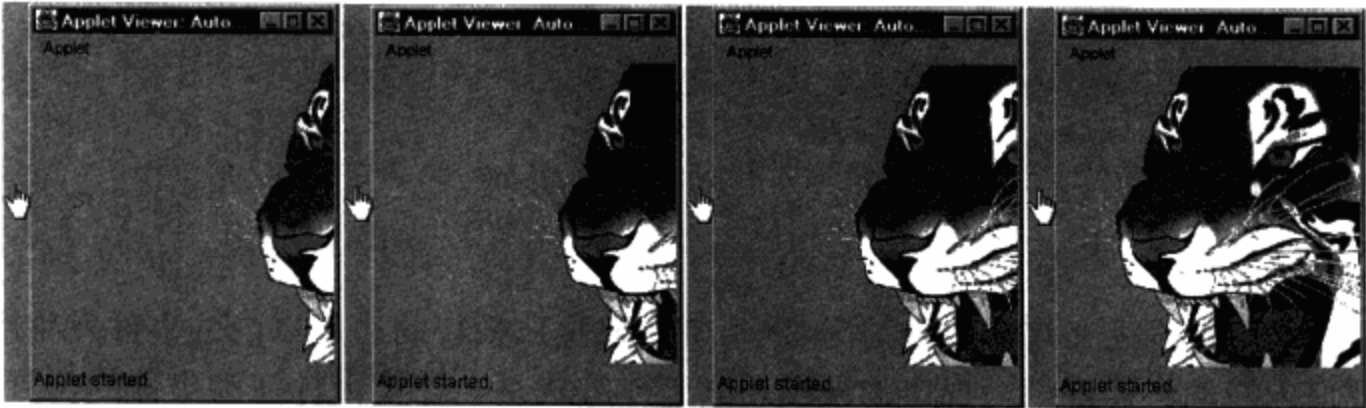


图 4-19 在定制组件中自动滚动

注意 JViewport 类只包含一个称作视口视图的组件。JViewport 为设置它的视图的位置提供了支持，很方便用来说明自动滚动。JViewport 组件在第 13.1 节“JViewport”中有详细的介绍。

例 4-16 列出了图 4-19 所示的小应用程序的代码。

例 4-16 自动滚动测试小应用程序

```
import javax.swing.* ;

public class AutoscrollTest extends JApplet {
    public AutoscrollTest () {
        JLabel label = new JLabel (new ImageIcon ("pic.gif"));
        JViewport vp = new AutoscrollViewport (label, 3);
        getContentPane ().add (vp, "Center");
    }
}
```

AutoscrollViewport 类的构造方法以一个组件和一个 integer 值为参数，这个组件是显示老虎

^① AutoscrollViewport 不是 Swing 版本的一部分。

图像的一个 JLabel 实例，integer 值代表滚动增量（以像素点为单位）。这个构造方法指定该组件作为视口的视图、启用自动滚动并把光标设置为预先定义的手形光标。

```
class AutoscrollViewport extends JViewport {
    point scrollTo = new Point(), last = new Point();
    boolean manualDragUnderway = false;
    final int increment;

    public AutoscrollViewport (Component component, int inc) {
        this.increment = inc;
        setView (component);
        setAutoscrolls (true);
        setCursor (Cursor.getPredefinedCursor (Cursor.HAND_CURSOR));
        ...
    }
}
```

鼠标事件的处理是通过两个内部类来完成的。当一个鼠标按下事件发生时，鼠标按下的位置被记录并设置了一个标志，指示手工拖动正在进行中。

```
...
addMouseListener (new MouseAdapter () {
    public void mousePressed (MouseEvent e) {
        last.x = e.getPoint().x;
        last.y = e.getPoint().y;
        manualDragUnderway = true;
    }
});
...
```

鼠标拖动事件发生在这样两种情况之一：事件发生在视口的边界以内或事件发生在视口外。

在视口内拖动鼠标将拖动视口的视图。其工作方式是：计算鼠标拖动事件的位置与前一个鼠标事件的位置之间的偏移。从当前视图的位置减去该偏移，得出的结果用于重新设置视口视图的位置。

```
...
addMouseMotionListener (new MouseMotionAdapter () {
    public void mouseDragged (MouseEvent e) {
        Point drag = e.getPoint();
        Point viewPos = getViewPosition();
        Point offset = new Point (drag.x-last.x,
                                   drag.y-last.y);

        last.x = drag.x;
        last.y = drag.y;
        if (contains (drag)) {
            if (manualDragUnderway) {
                scrollTo.x = viewPos.x-offset.x;
                scrollTo.y = viewPos.y-offset.y;
                setViewPosition (scrollTo);
            }
        }
    }
});
...
```

如果鼠标拖动事件没有发生在视口的范围内，则视口通过向鼠标方向增量移动视图来实现自动滚动。

根据鼠标在视口的上面、下面、左边或右边，视口重新设置其视图及标签的位置。根据传送给 AutoscrollViewport 构造方法的增量值，视图的位置在水平或垂直方向上进行调整。

```

...
else { // autoscrolling...
    Rectangle bounds = getBounds();
    ...
    if (drag.x > bounds.x + bounds.width) {
        // scroll right
        viewPos.x -= increment;
        setViewPosition (viewPos);
    }
    if (drag.x < 0) {
        // scroll left
        viewPos.x += increment;
        setViewPosition (viewPos);
    }
    if (drag.y > bounds.y + bounds.height) {
        // scroll down
        viewPos.y -= increment;
        setViewPosition (viewPos);
    }
    if (drag.y < 0) {
        // scroll up
        viewPos.y += increment;
        setViewPosition (viewPos);
    }
}
});
}

```

只要光标在视口外且鼠标按钮按下，视图就会向鼠标方向滚动。上面的实现自动滚动的代码是简化了的，工业版本的自动滚动在视图刚滚出视口时将停止自动滚动。

例 4-17 列出了 AutoscrollViewport 的完整代码。

例 4-17 AutoscrollViewport 列表

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class AutoscrollViewport extends JViewport {
    Point scrollTo = new Point(), last = new Point();
    boolean manualDragUnderway = false;
    final int increment;

    public AutoscrollViewport (Component component, int inc) {
        this.increment = inc;
        setView (component);
        setAutoscrolls (true);
        setCursor (Cursor.getPredefinedCursor (Cursor.HAND_CURSOR));
        addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                last.x = e.getPoint().x;
            }
        });
    }
}

```

```
        last.y = e.getpoint().y;
        manualDragUnderway = true;
    }
};

addMouseListener (new MouseMotionAdapter () {
    public void mouseDragged (MouseEvent e) {
        point drag = e.getPoint();
        Point viewPos = getViewposition();
        Point offset = new Point (drag.x - last.x,
                                   drag.y - last.y);

        last.x = drag.x;
        last.y = drag.y;

        if (contains (drag)) {
            if (manualDragUnderway) {
                scrollTo.x = viewPos.x - offset.x;
                scrollTo.y = viewPos.y - offset.y;
                setViewPosition (scrollTo);
            }
            else { // autoscrolling ...
                Rectangle bounds = getBounds();
                manualDragUnderway = false;

                if (drag.x > bounds.x + bounds.width) {
                    // scroll right
                    viewPos.x -= increment;
                    setViewPosition (viewPos);
                }
                if (drag.x < 0) {
                    // scroll left
                    viewPos.x += increment;
                    setViewPosition (viewPos);
                }
                if (drag.y > bounds.y + bounds.height) {
                    // scroll down
                    viewPos.y -= increment;
                    setViewPosition (viewPos);
                }
                if (drag.y < 0) {
                    // scroll up
                    viewPos.y += increment;
                    setViewPosition (viewPos);
                }
            }
        }
    }
});
```

4.7 工具提示

工具提示是当鼠标停留在一个组件上的时间超出指定的时间后在一个窗口中显示的一行文字。

把工具提示与一个 Swing 组件相关联是通过调用 `JComponent.setToolTipText()` 来实现的。`JComponent.setToolTipText()` 所带的参数是将在工具提示窗口中显示的字符串。把 `null` 字符串传送给 `setToolTipText()` 将使该组件以前的工具提示无效。

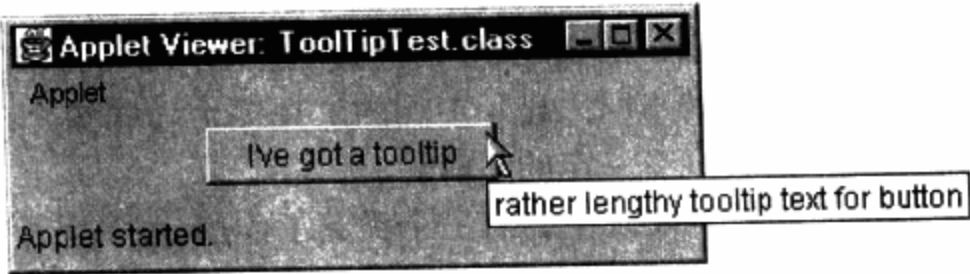


图 4-20 显示在一个单独的窗口中的工具提示

图 4-20 示出了一个与一个 `JButton` 实例相关联的工具提示。

该按钮的工具提示设置为一个相当长的字符串，用以说明工具提示是显示在它们自己的窗口中的（这个工具提示的窗口超出了小应用程序窗口的边框）。然而，好的 GUI 设计规定工具提示文本应该是言简意赅的，一般不要超过数个字。因此，工具提示不支持多行文本。例 4-18 列出了图 4-20 所示的小应用程序的代码。

例 4-18 为一个按钮设置工具提示文本

```
import javax.swing.*;
import java.awt.*;

public class ToolTipTest extends JApplet {
    public void init() {
        Container contentPane = getContentPane();
        JButton button = new JButton("I've got a tooltip");

        button.setToolTipText(
            "rather lengthy tooltip text for button");

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);
    }
}
```

该小应用程序实例化一个 `JButton` 实例，为按钮设置工具提示文本并把按钮添加到小应用程序的内容窗格中。当光标停留在该按钮上一定时间后，就会显示工具提示。显示工具提示的初始时间延迟是可设置的，这将在 4.7.3 节“定制工具提示的行为”中介绍。

`JComponent` 类为设置工具提示和定制它们的行为提供了许多选项，如设置基于鼠标位置的工具提示文本。表 4-2 列出了处理工具提示的 `JComponent` `public` 方法。

表 4-2 处理工具提示的 JComponent public 方法

方法名	实现
<code>setToolTipText</code>	带一个用作工具提示文本的字符串
<code>getToolTipText</code>	无参数——返回传送给 <code>setToolTipText()</code> 的字符串
<code>getToolTipText</code>	带一个鼠标事件——返回由 <code>setToolTipText()</code> 设置的字符串 对与鼠标位置有关的工具提示可以在扩展中重载这个方法
<code>getToolTipLocation</code>	返回一个基于鼠标位置、代表工具提示文本首选位置的点坐标
<code>createToolTip</code>	返回被组件使用的工具提示。可为要求定制工具提示的组件重载该方法

我们已介绍了 `SetToolTipText` 方法，表 4-2 所列的其他方法将在下面的介绍。

4.7.1 基于鼠标位置的工具提示

与大多数 Swing 特性一样，与工具提示有关的特性也相当复杂。然而，正如例 4-18 所列的

小应用程序所说明的那样，复杂性已经隐藏了，大多数开发人员不用去处理。如果只需为组件显示一个工具提示，那么可以只调用该组件的 `setToolTipText` 方法。然而，如果希望在单个组件上根据鼠标位置的不同来显示不同的工具提示（例如，在一个图像映像上显示信息），则需要对工具提示的实现有一定的了解。

在工具提示的实现上，有三个类起了作用。

第一是 `JComponent` 类，如前所述，`JComponent` 类提供了一个 `setToolTipText` 方法，该方法为组件激活工具提示并设置与工具提示有关的文本。

第二是 `JToolTip` 类，由于 Swing 插入式界面样式的实现，这个类仅维护工具提示文本和一个对与工具提示有关的组件的引用。实际显示工具提示文本的代码封装在工具提示的界面样式中，而界面样式又存在于 `swing.plaf.ToolTipUI` 类的扩展中。

第三，Swing 的 `ToolTipManager` 类负责提供一个将在其中显示工具提示的窗口，并负责把 `JToolTips` 实例插入到显示它们的窗口中。`ToolTipManager` 还控制定时问题，如鼠标停留在组件上的时间与显示工具提示的时间之间的时间差。

`JComponent` 提供 `getToolTipText()` 的两个版本：一个无参数的版本，它只返回传送给 `setToolTipText()` 的字符串，另一个版本则带一个 `MouseEvent` 参数。

当显示工具提示时，Swing 的 `ToolTipManager` 调用 `JComponent.getToolTipText(MouseEvent)` 来获得工具提示的文本。`JComponent.getToolTipText(MouseEvent)` 忽略 `MouseEvent`，只返回由 `JComponent.getToolTipText()` 返回的字符串。结果，不管光标的位置如何，总是返回通过调用 `setToolTipText()` 而设置的文本。

很明显，`JComponent.getToolTipText(MouseEvent)` 只是为了被 `JComponent` 的扩展重载而设计的。定制组件可以重载 `getToolTipText(MouseEvent)` 以返回基于光标位置的文本，如图 4-21 所示。



图 4-21 基于光标位置的工具提示

例 4-19 列出了图 4-21 所示的小应用程序的代码。

例 4-19 根据鼠标位置来显示不同的工具提示

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ToolTipsBasedOnMousePosition extends JApplet {
    public void init() {
        Container contentPane = getContentPane();
        ImageMap map = new ImageMap("tiger.gif");
        contentPane.setLayout(new FlowLayout());
    }
}
```

```

        contentPane.add (map);
    }
}
class ImageMap extends JLabel {
    private Rectangle teeth = new Rectangle (62, 203, 80, 55),
        nose = new Rectangle (37, 164, 130, 30),
        ear = new Rectangle (228, 10, 65, 55),
        rEye = new Rectangle (137, 103, 20, 17),
        lEye = new Rectangle (65, 97, 16, 15);

    public ImageMap (String imageName) {
        super (new ImageIcon (imageName));
        setToolTipText ("tiger!");
    }

    public String getToolTipText (MouseEvent e) {
        Point p = e.getPoint ();
        String s = null;

        if (teeth.contains (p)) s = "oooooh, big teeth!";
        else if (nose.contains (p)) s = "keen sense of smell";
        else if (ear.contains (p)) s = "acute hearing";
        else if (rEye.contains (p) || lEye.contains (p))
            s = "excellent vision";

        return s == null ? getToolTipText (): s;
    }
}

```

为了返回光标位置处的文本，ImageMap 类扩展 JLabel 并重载 getToolTipText()。如果光标不在目标位置上，则重载的 getToolTipText() 方法通过调用 JComponent.getToolTipText() 来模仿 JComponent.getToolTipText(MouseEvent) 的实现。

JComponent.setToolTipText(MouseEvent) 除设置工具提示文本外，还向 ToolTipManager 登记组件。因此，虽然 ImageMap 类重载了 getToolTipText(MouseEvent) 方法，但是，要显示工具提示还必须调用 ImageMap 构造方法中的 setToolTipText() 方法。

4.7.2 工具提示的首选位置

重载 JComponent.getToolTipLocation 方法可以用来设置工具提示的首选位置。getToolTipLocation 方法带一个 MouseEvent 并返回一个 Point，工具提示将显示在 Point 的位置上。缺省时，JComponent.getToolTipLocation() 返回 null，指示工具提示应该直接显示在鼠标光标的下面。JComponent 的扩展可以重载 getToolTipText() 来指定要显示的工具提示的确切位置。

例如，例 4-19 所列的小应用程序通过重载 getToolTipLocation 可以指定图像映像的工具提示的位置。

```

class ImageMap extends JLabel {
    private Rectangle teeth = new Rectangle (62, 203, 80, 55),
        nose = new Rectangle (37, 164, 130, 30),
        ear = new Rectangle (228, 10, 65, 55),
        rEye = new Rectangle (137, 103, 20, 17),
        lEye = new Rectangle (65, 97, 16, 15);

    ...

    public Point getToolTipLocation (MouseEvent e) {
        if (teeth.contains (new Point (e.getX(), e.getY())))

```



```
return new Point (teeth.x, teeth.y);
```

如果光标停留在老虎的牙齿上，则指定老虎嘴的左上角为工具提示的位置。不像图 4-21 左图那样工具提示显示在光标下面，这里的工具提示如图 4-22 所示。注意光标的位置与工具提示位置之间的关系。

可在 10.10 节“JToolBar”中找到另一个重载 `getToolTipLocation` 的例子。

4.7.3 定制工具提示的行为

如前所述，当光标在组件上停留一段时间后就会显示工具提示，这里的一定时间指的是初始延迟。缺省时，初始延迟被设置为 750 毫秒。

此后，下述三个事件之一的事件可清除工具提示：

- 光标离开了这个组件。
- 按下了鼠标。
- 清除延迟结束。

清除延迟是分配给读者阅读工具提示的时间，如 4 秒，它提供光标停留在组件边界上并且不按下鼠标的时间。

如果光标离开显示工具提示的组件，接着进入另一个带工具提示的组件，则第二个组件的工具提示可能立即显示出来。如果退出第一个组件进入第二个组件之间的时间少于重新显示延迟（`reshow delay`），则第二个组件的工具提示将会立即显示。缺省的重新显示延迟时间是 500 毫秒。表 4-3 总结了工具提示的延迟。

表 4-3 工具提示的延迟		
延迟	缺省值	含义
初始	750 毫秒	从光标停在组件上到工具提示显示之间的延迟
清除	4 秒	从显示工具提示到清除工具提示之间的延迟
重新显示	500 毫秒	如果光标离开了显示工具提示的组件并在重新显示延迟终止之前进入了另一个带工具提示的组件，则第二个组件的工具提示立即显示

工具提示延迟是由 `ToolTipManager` 类控制的，该类的一些方法设置表 4-3 所列的延迟。由 `ToolTipManager` 提供的 `public` 方法是开发人员感兴趣的方法，这些方法列在表 4-4 中。设置和获取延迟的所有方法都以整数为参数。

通过调用 `ToolTipManager` 的 `setEnabled()` 方法（该方法以一个 `boolean` 值为参数），可以为所有组件启用或禁用工具提示。如果该方法载入的参数是 `true`，则所有带工具提示文本的组件都可以使用工具提示。如果该方法载入的参数是 `false`，则所有组件都将禁用工具提示。



图 4-22 在指定位置上显示的工具提示

表 4-4 `ToolTipManager` 的 `public` 方法

方法名	实现
<code>setInitialDelay</code>	以毫秒为单位设置初始延迟
<code>getInitialDelay</code>	以毫秒为单位返回初始延迟
<code>setDismissDelay</code>	以毫秒为单位设置清除延迟
<code>getDismissDelay</code>	以毫秒为单位返回清除延迟
<code>setReshowDelay</code>	以毫秒为单位设置重新显示延迟
<code>getReshowDelay</code>	以毫秒为单位返回重新显示延迟
<code>setEnabled</code>	启用/禁用所有的工具提示
<code>isEnabled</code>	返回所有工具提示的启用状态

Swing 提示

工具提示是如何工作的

当光标停留在带一个工具提示的组件上时，计时器开始工作，缺省时，它运行 750 毫秒。如果在计时器运行期间，光标没有移动或鼠标按钮没有按下，则 750 毫秒之后，显示工具提示。该延迟称作初始延迟。

当工具提示显示时，第二个计时器（缺省时运行 4 秒）开始工作。如果鼠标没有移出组件并且鼠标没有被按下，则工具提示将一直显示直到用完 4 秒。该延迟称作清除延迟。

当光标退出正显示工具提示的组件时，另一个计时器开始工作，缺省时它运行 0.5 秒。如果光标在该计时器运行完之前，就进入了第二个带工具提示的组件，则第二个组件的工具提示立即显示。该延迟称作重新显示延迟。这三种延迟都可以通过 `ToolTipManager` 的静态方法来设置。

4.7.4 定制工具提示的界面样式

与大多数 Swing 组件一样，工具提示也有一个 UI 代表，该 UI 代表决定该工具提示的界面样式。例如，图 4-23 显示了两个都显示工具提示但具有不同的界面样式的小应用程序，左图的小应用程序具有 Macintosh 界面样式，右图的小应用程序则配备了一个 organic 界面样式。

JComponent 类负责创建它自己的工具提示。工具提示和工具提示管理器完成工具提示的显示，组件则负责创建工具提示。

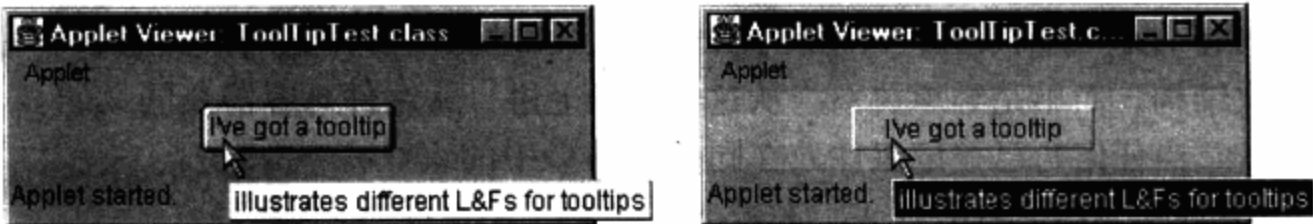


图 4-23 有不同界面样式的工具提示

`JComponent.createToolTip()` 实例化一个 `JToolTip` 实例并返回该实例。极少数情况下可以重载 `createToolTip` 方法，如果需要为定制的组件指定工具提示的类型的话。例如，如果需要在工具提示中使用多行文本，则应该扩展 `JToolTip` 类以提供这种功能。此后，需要多行工具提示的组件可以重载它们的 `createToolTip` 方法以返回多行 `JToolTip` 扩展的一个实例。然而，实际中，很少重载 `JComponent.createToolTip`。

4.8 键击处理

`JComponent` 类为处理嵌套组件的键击提供了一些措施。表 4-5 列出了与键击有关的 `JComponent` 方法。

`JComponent.registerKeyboardAction (ActionListener, KeyStroke, int)` 登记一个键击；当该键击发生时，如果满足 `registerKeyboardAction ()` 的 `integer` 参数指定的条件，则调用 `ActionListener` 的 `actionPerformed` 方法。

调用 `unregisterKeyboardAction ()` 可以注销键击。可以通过分别调用 `getActionForKeyStroke ()` 和 `getConditionForKeyStroke ()` 来获得 `ActionListener` 和与一个已登记的键击有关的条件。

`getRegisteredKeyStroke` 方法返回与一个组件有关的所有键击，调用 `resetKeyboardActions ()` 可以清除一个组件的键盘动作。

键击由 `KeyStroke` 的一个实例来指定，这个实例封装一个键、`Ctrl`、`Alt` 或 `Shift` 等功能键，

以及该键击是否与键按下或键释放有关。

由传送给 registerKeyboardAction () 的 integer 常量指定处理键击的条件。表 4-6 列出了这些常量。

表 4-5 JComponent 的键击处理方法

方法名	实现
registerKeyboardAction	传送一个 ActionListener、KeyStroke 和一个整数条件，向组件登记这个键击
unregisterKeyboardAction	传送一个要注销的键击
getActionForKeyStroke	返回与一个 KeyStroke 相关联的动作
getConditionForKeyStroke	为指定的 KeyStroke 返回整数代表条件
getRegisteredKeyStrokes	返回一个已登记的键击的数组
resetKeyboardActions	清除组件的所有键盘动作

表 4-6 键击处理的 JComponent 常量

常量	含义
WHEN _ FOCUSED	当已登记的组件有焦点时处理键击
WHEN _ ANCESTOR _ OF _ FOCUSED _ COMPONENT	当已登记的组件有焦点或已登记的组件是有 _ COMPONENT 焦点的组件的父组件时处理键击
WHEN _ IN _ FOCUSED _ WINDOW	当已登记的组件有焦点或当与已登记的组件在同一个窗口的其他组件有焦点时处理键击

当向一个特定的组件登记了键击时，这个组件就称作已登记的组件。处理一个键击指的是与这个键击相关联的 ActionListener 调用其 actionPerformed 方法。

当已登记的组件有焦点时，处理满足 WHEN _ FOCUSED 条件的键击。

当已登记的组件有焦点或已登记的组件包含有焦点的组件时，则处理满足 WHEN _ ANCESTOR _ OF _ FOCUSED _ COMPONENT 条件的键击。对键击处理而言，术语父组件指容器包含层次结构，而不是继承关系。从这个意义上讲，父组件与容器是同义的。

当任何与已登记组件处在同一个窗口中的组件有焦点时，处理 WHEN _ IN _ FOCUSED _ WINDOW 键击。

图 4-24 显示了一个小应用程序，它包含两个由一个 BorderLayout 实例来布局的组件：一个面板（作为中心组件）和一个按钮（作为南边组件）。该面板又包含一个按钮和一个复选框。

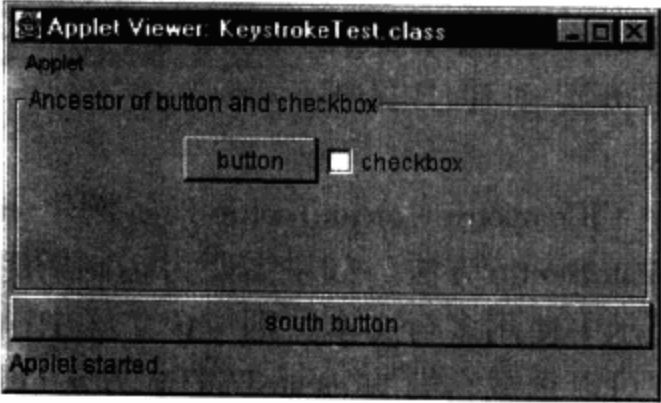


图 4-24 嵌套组件中的键击处理

在创建了小应用程序中的组件后，就向这个面板、复选框和 South button 按钮登记键击。

首先，向复选框登记一个键击。KeyStrokes 是唯一的而且是不变的，获得对 KeyStrokes 的一个引用的唯一办法是使用 static KeyStroke.getKeyStroke 方法，该方法从内存返回一个 KeyStrokes。

与复选框相关联的键击是用一个不带修饰键、按下的“f”（表示焦点的意思）键来表示的。当已登记组件（在这种情况下是复选框）有焦点时，处理该键击。通过调用 KeyStroke.getKeyStroke () 来获得键击，KeyStroke.getKeyStroke 有三个参数：一个代表这个键的常量 (KeyEvent.VK _ F，它来自 java.awt.event.KeyEvent 类)、与该键按下有关的修饰键 (0)、以及这个键击是代表键按下 (false) 还是代表键释放 (true)。

```
public class keystrokeTest extends JApplet {
    ...
    public void init () {
        ...
        Listener listener = new Listener ();
        ...
    }
}
```

```
checkbox.registerKeyboardAction (
    listener,
    KeyStroke.getKeyStroke (KeyEvent.VK_F, 0, false),
    JComponent.WHEN_FOCUSED);
...
```

在这个小应用程序中的所有键击都指定 Listener 的一个实例作为 ActionListener，它的 actionPerformed 方法在处理键击时被调用。Listener.actionPerformed() 确定与键击相关联的已登记组件并显示一个恰当的消息。

```
class Listener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        Object src = e.getSource();
        String cname = src.getClass().getName();

        if (src instanceof JCheckBox) {
            System.out.print ("'" + e.getKeyChar() + "' key PRESSED when checkbox");
            System.out.println ("had focus");
        }
        else if (src instanceof JPanel) {
            System.out.print ("'" + e.getKeyChar() + "' key PRESSED when a");
            System.out.println ("component contained in the");
            System.out.println ("titled panel had focus");
        }
        else if (src instanceof JButton) {
            System.out.print ("'" + e.getKeyChar() + "' key RELEASED when any");
            System.out.println ("component in window had focus");
        }

        System.out.println ("Source: " + cname);
        System.out.println ();
    }
}
```

结果，当复选框有焦点时，按下“f”键会产生下面的输出：

```
'f' key PRESSED when checkbox had focus
source: javax.swing.JCheckBox
```

接着，向面板登记一个键击。该键击用 Alt 修饰键加按下的“a”（表示父组件）键来表示。

```
...
panel.registerKeyboardAction (
    listener,
    keyStroke.getKeyStroke (KeyEvent.VK_A,
        InputEvent.ALT_MASK, false),
    JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
...
```

当面板是具有焦点的组件的父组件时，处理这个键击，这是说当面板中的按钮或复选框有焦点时，就处理这个键击。当已登记的组件（在这种情况下是面板）有焦点但 JPanel 类不接受键盘焦点时，处理满足 WHEN_ANCESTOR_OF_FOCUSED_COMPONENT 条件的键击。

如果面板中的按钮或复选框有焦点的话，按下 Alt + a 会产生下面的输出：

```
'ALT-a' key PRESSED when a component contained in the titled
panel had focus Source: javax.swing.JPanel
```

最后，向直接添加到这个小应用程序的内容窗格中的按钮登记键击。该键击由不带修饰键

的、释放的“w”键来表示。当同一窗口中的已登记的组件或该按钮等组件有焦点时，处理这个键击。

```
...
southButton.registerKeyboardAction (
    listener,
    keyStroke.getKeyStroke (KeyEvent.VK_W, 0, true),
    JComponent.WHEN_IN_FOCUSED_WINDOW);
...
```

当窗口中的任意组件有焦点且“w”键释放时，输出如下信息：

```
'w' key RELEASED when any component in window had focus
Source: javax.swing.JButton
```

总结一下，图 2-24 所示的小应用程序中处理键击的情况是这样的：如果面板包含的按钮有焦点时，则会处理按下 Alt + a 键和释放“w”键。如果面板包含的复选框有焦点时，当适当的键被按下或释放时，这三种键击都会被处理。如果 South Button 按钮有焦点时，当释放“w”键时，只有“w”键击被处理。

注意 在 Macintosh 界面样式中，该小应用程序将不会工作，因为 Macintosh 按钮不能接受焦点。

例 4-20 列出了该小应用程序的代码。

例 4-20 处理嵌套键击

```
import java.applet.Applet;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KeystrokeTest extends JApplet {
    private JButton button = new JButton ("button");

    public void init () {
        Container contentPane = getContentPane ();
        JPanel panel = new JPanel ();
        JCheckBox checkbox = new JCheckBox ("checkbox");
        JButton southButton = new JButton ("south button");
        Listener listener = new Listener ();

        panel.setBorder (
            BorderFactory.createTitledBorder (
                ("Ancestor of button and checkbox")));

        checkbox.registerKeyboardAction (
            listener,
            KeyStroke.getKeyStroke (KeyEvent.VK_F, 0, false),
            JComponent.WHEN_FOCUSED);

        panel.registerKeyboardAction (
            listener,
            KeyStroke.getKeyStroke (KeyEvent.VK_A,
                InputEvent.ALT_MASK),
            JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);

        southButton.registerKeyboardAction (
            listener,
```

```
        KeyStroke.getKeyStroke (KeyEvent.VK_W, 0, true),
        JComponent.WHEN_IN_FOCUSED_WINDOW);

    panel.add (button);
    panel.add (checkbox);

    contentPane.add (panel, "Center");
    contentPane.add (southButton, "South");
}

class Listener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        Object src = e.getSource ();
        String cname = src.getClass ().getName ();

        if (src instanceof JCheckBox) {
            System.out.print ("f' key PRESSED when checkbox");
            System.out.println (" had focus");
        }
        else if (src instanceof JPanel) {
            System.out.print ("ALT-a' key PRESSED when a");
            System.out.println ("component contained in the");
            System.out.println (" titled panel had focus");
        }
        else if (src instanceof JButton) {
            System.out.print ("w' key RELEASED when any");
            System.out.println (" component in window had focus");
        }

        System.out.println ("Source: " + cname);
        System.out.println ();
    }
}
```

4.9 客户属性

每个 JComponent 实例都维护一个称作客户属性的属性字典。一个字典由一组键/值对组成,这里,键和值可能是任何类型的对象。Swing 类本身使用客户属性。例如,组件把它们的工具提示文本作为一个客户属性来维护,JLayeredPane 类^①把它的组件层作为一个客户属性来维护。

表 4-7 客户属性的 JComponent public 方法

方法名	实现
putClientProperty	传送一个键和一个值,它们都是类型对象。把该值与一个已存在的客户属性相关联或创建具有指定键/值对的新客户属性
getClientProperty	返回与指定的客户属性键(一个对象)相关联的值(一个对象)

客户属性是被约束的属性,即不论何时一个客户属性修改了,那么属性变化事件就会发送给组件的属性变化监听器。表 4-7 列出了处理客户属性的 JComponent 方法。

维护客户属性的字典实际上是 JComponent 类的一个 private 成员,而不是 public(或 protected)访问方法。因此,获得为给定组件设置的所有客户属性的详细信息是不可能的。表 4-7 所列的方法只能访问个别属性。

putClientProperty 方法允许一个键/值对作为客户属性登记。如果 null 值传送给 putClientProperty(), 并且具有特定键的客户属性存在的话,则该属性就从客户属性字典中清除。

① JLayeredPane 是一个容器,它可在单独的层上显示其组件。

`getClientProperty` 方法返回与客户属性（由指定关键字确定）相关联的值。如果没有这样的客户属性，则返回一个 `null` 值。

图 4-25 所示的小应用程序包含一个维护动态目标的按钮。当该按钮被激活时，它把组件的背景颜色指定为它的“目标”的客户属性。

该按钮的目标是在这个小应用程序显示的三个面板中的一个面板，通过从组合框中选取一个选项来设置目标。

这个小应用程序创建按钮、组合框和三个面板。

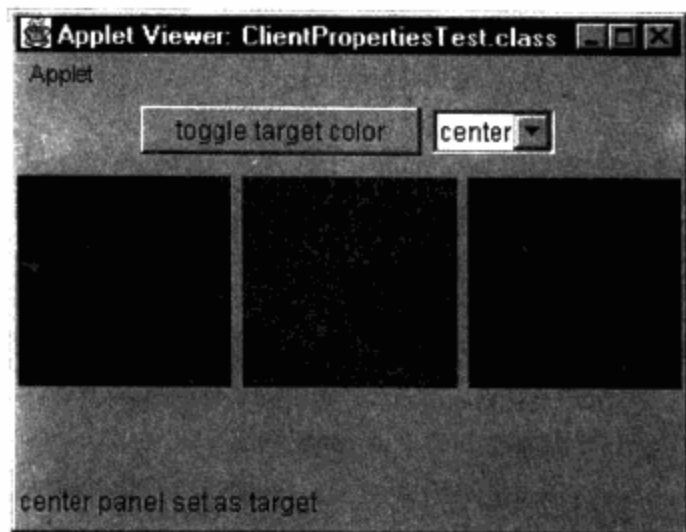


图 4-25 具有一个动态目标的按钮

```
public class Test extends JApplet {
    JButton button = new JButton ("toggle target color");
    JComboBox targetCombo = new JComboBox ();
    JPanel [] targets = { new JPanel (),
                          new JPanel (),
                          new JPanel () };
    ...
}
```

这些目标是 `JPanel` 的实例，这些实例把它们的首选大小设置为 100 像素宽和 100 像素高。当组合框激活时，把按钮的目标设置为所选的选项。

```
targetCombo.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        button.putClientProperty (
            "target",
            targets [targetCombo.getSelectedIndex ()]);
    }
});
```

按钮的动作监听器以“target”客户属性的方式获得了对组件的一个引用。然后设置了目标的背景并重新绘制了目标。

```
button.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        Component c =
            (Component) button.getClientProperty ("target");
        if (c != null) {
            color bg = c.getBackground ();
            c.setBackground (bg == Color.blue ?
                             Color.red : Color.blue);
            c.repaint ();
        }
    }
});
```

这个小应用程序把一个属性变化监听器添加到按钮中。当“target”客户属性修改时，监听器就相应地更新小应用程序的状态条。

```
button.addPropertyChangeListener (
    new PropertyChangeListener () {
        public void propertyChange (PropertyChangeEvent e) {
            if (e.getPropertyName ().equals ("target")) {
```

```

        showStatus (
            (String) targetCombo.getSelectedItem() +
            " panel set as target");
    }
}
);

```

例-21 列出了图 4-25 所示的小应用程序的完整代码。

例 4-21 用客户属性把一个动态目标分配给一个按钮

```

import javax.swing. * ;
import java.util. * ;
import java.awt. * ;
import java.awt.event. * ;
import java.beans. * ;

public class ClientPropertiesTest extends JApplet {
    JButton button = new JButton ("toggle target color");
    JComboBox targetCombo = new JComboBox ();
    JPanel [] targets = {new JPanel (),
                        new JPanel (),
                        new JPanel ()};

    public void init () {
        Container contentPane = getContentPane ();
        Dimension targetPreferredSize = new Dimension (100, 100);
        JPanel targetPanel = new JPanel ();

        for (int i = 0; i < targets.length; ++ i) {
            targets [i].setBackground (Color.blue);
            targets [i].setPreferredSize (targetPreferredSize);
            targetPanel.add (targets [i]);
        }

        targetCombo.addItem ("left");
        targetCombo.addItem ("center");
        targetCombo.addItem ("right");

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);
        contentPane.add (targetCombo);
        contentPane.add (targetPanel);

        button.putClientProperty ("target", targets [0]);
        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                Component c =
                    (Component) button.getClientProperty ("target");

                if (c != null) {
                    Color bg = c.getBackground ();
                    c.setBackground (bg == Color.blue ?
                                    Color.red : Color.blue);

                    c.repaint ();
                }
            }
        });

        targetCombo.addActionListener (new ActionListener () {

```

```
public void actionPerformed (ActionEvent e) {
    button.putClientProperty (
        "target",
        targets [targetCombo.getSelectedIndex ()]);
}
button.addPropertyChangeListener (
    new PropertyChangeListener () {
        public void propertyChange (PropertyChangeEvent e) {
            if (e.getPropertyName ().equals ("target")) {
                showStatus (
                    (String) targetCombo.getSelectedItem () +
                    " panel set as target");
            }
        }
    });
}
```

Swing 提示

客户属性与继承的比较

客户属性使 JComponent 类有一定程度的扩展性，否则，这种扩展性只能通过子类化来获得。例如，为了不把一个组件子类化和维护对子类中的一个对象的引用，可把客户属性添加到原来的组件中。这种可能性在定制组件行为时带来了很大的灵活性并且减少了继承的使用。

4.10 焦点管理

缺省时，在 Swing 容器中按下 Tab 键将把焦点移到下一个可获得焦点的组件上，按下 Shift + Tab 可把焦点移到上一次获得焦点的组件上。可以用 JComponent 的焦点属性或缺省焦点管理器的替代属性来修改缺省的行为。

4.10.1 JComponent 的焦点属性

所有的轻量 Swing 组件维护五个与表 4-8 所列的焦点管理有关的属性。表 4-8 中设置方法一栏指出 JComponent 类是否提供设置该属性的方法。未提供 focusCycleRoot、focusTraversable 和 managingFocus 属性的设置方法。因此，要修改这些缺省的行为，必须在 JComponent 的一个扩展中重载 JComponent 的方法 isFocusCycleRoot ()、isFocusTraversable ()和 isManagingFocus ()。

表 4-8 JComponent 焦点属性

属性名	数据类型	设置方法	缺省
focusCycleRoot	boolean		false
focusTraversable	boolean		true
managingFocus	boolean		false
requestFocusEnabled	boolean		true
nextFocusableComponent	Component		null

focusCycleRoot——确定一个容器是否包含形成它们自己的焦点循环的那些组件。如果该属

性设置为 true，则按下 Tab 键将把焦点移到该容器中并且焦点会在容器的组件中移动，但不会移出容器本身。

focusTraversable——确定焦点管理器是否将把焦点传给一个组件。其 **focusTraversable** 属性指定为 false 的组件在 **requestFocus()** 被调用时仍将接受焦点，例如，如果一个按钮的 **focusTraversable** 属性被设置为 false（通过在 **JButton** 的一个扩展中重载 **isFocusTraversable()**），则单击该按钮会产生对 **requestFocus()** 的一个调用，并且该按钮仍将接受焦点。但是，如果按下 Tab 或 Shift + Tab，则该按钮将不接受焦点。

managingFocus——确定当按下键把焦点切换到一个组件或从一个组件把焦点移开时，是否要把所按下的键传送给这个组件本身。缺省时是不传送的。

requestFocusEnabled——与 **focusTraversable** 属性相反，如果该属性设置为 false，则在 **requestFocus()** 调用时组件将不接受焦点，但是，作为键按下的结果，焦点管理器将把焦点传给该组件。

nextFocusableComponent——在按下 Tab 键时，指定接受焦点的下一个组件。注意，不能把焦点传送给以前的可获得焦点的组件。

图 4-26 所示的小应用程序图解说明了设置列表 4-8 中的属性。

左上按钮是 **JButton** 的一个扩展，它重载 **isFocusTraversable** 以返回 false。如果该按钮是激活的，则它将接受焦点；然而，按下 Tab 或 Shift + Tab 时，该组件将不接受焦点。

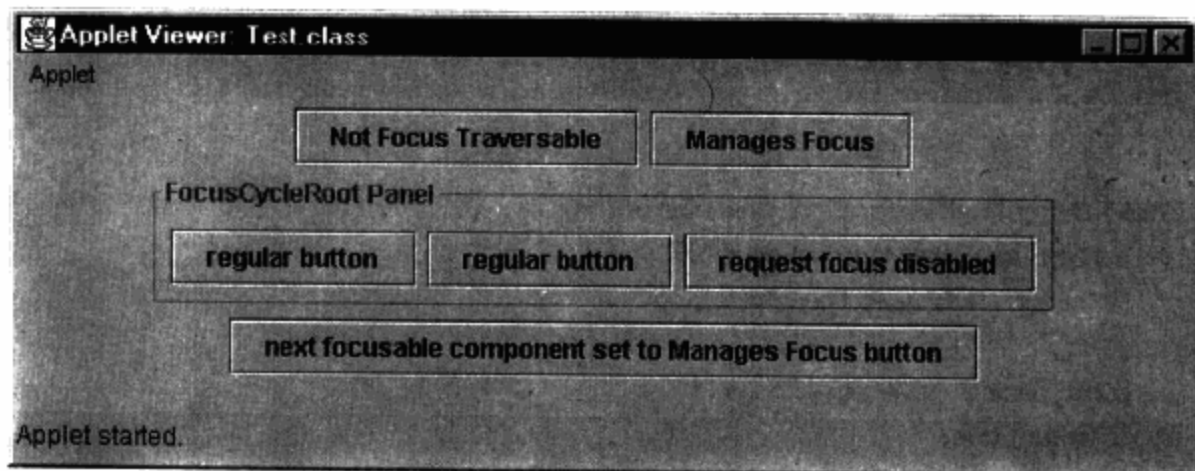


图 4-26 为 Swing 组件指定焦点属性

右上按钮也是 **JButton** 的一个扩展，它重载 **isManagingFocus** 以返回 false。当该按钮作为按下 Tab 或 Shift + Tab 的结果接受焦点时，键按下被传送给该组件的 **processComponentKeyEvent** 方法。

“**FocusCycleRoot Panel**”是 **JPanel** 的一个扩展，它重载 **isFocusCycleRoot()** 以返回 true。一旦把焦点给面板中的一个按钮时，按下 Tab 或 Shift + Tab 将不会把焦点移出面板外，但将在面板中的按钮间循环。

“request focus disabled”按钮替自己调用 **setRequestFocusEnabled(false)**。当 **requestFocus()** 被调用（例如，当该按钮被激活时），该按钮将不接受焦点，但是，当 Tab 或 Shift + Tab 键被按下时，该按钮将接受焦点。

最下面的组件指定下一个可获得焦点的组件是“**Manages Focus**”按钮。当该按钮有焦点时按下 Tab 将把焦点传送给“**Manages Focus**”按钮。

例 4-22 列出了图 4-26 所示的小应用程序的代码。

例 4-22 为 Swing 组件指定焦点属性

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Test extends JApplet {
```

```
private JButton button_1 = new NotFocusTraversableButton (),
    button_2 = new ButtonThatManagesFocus (),
    button_3 = new JButton ("regular button"),
    button_4 = new JButton ("regular button"),
    button_5 = new JButton ("request focus disabled"),
    button_6 = new JButton (
        "next focusable component set to Manages Focus button");

public void init () {
    Container contentPane = getContentPane ();
    FocusCycleRootPanel panel = new FocusCycleRootPanel ();

    button_5.setRequestFocusEnabled (false);
    button_6.setNextFocusableComponent (button_2);

    panel.add (button_3);
    panel.add (button_4);
    panel.add (button_5);

    contentPane.setLayout (new FlowLayout ());
    contentPane.add (button_1);
    contentPane.add (button_2);
    contentPane.add (panel);
    contentPane.add (button_6);
}

class ButtonThatManagesFocus extends JButton {
    public ButtonThatManagesFocus () {
        super ("Manages Focus");
    }

    public boolean isManagingFocus () {
        return true;
    }

    public void processComponentKeyEvent (KeyEvent e) {
        System.out.println (e);
    }
}

class NotFocusTraversableButton extends JButton {
    public NotFocusTraversableButton () {
        super ("Not Focus Traversable");
    }

    public boolean isFocusTraversable () {
        return false;
    }
}

class FocusCycleRootPanel extends JPanel {
    public FocusCycleRootPanel () {
        setBorder (BorderFactory.createTitledBorder (
            "FocusCycleRoot Panel"));
    }

    public boolean isFocusCycleRoot () {
        return true;
    }
}
```

4.10.2 焦点管理器

Swing 的焦点管理器负责把焦点从一个组件传送给另一个组件。Swing 提供了一个抽象的 `FocusManager` 类和一个 `DefaultFocusManager` 扩展。`DefaultFocusManager` 类把焦点从左传送到右，从上传送到下。

缺省的焦点管理器负责确定用哪个键来移动焦点。缺省时，`Tab` 和 `Ctrl + Tab` 向前移动焦点，`Shift + Tab` 和 `Ctrl + Shift + Tab` 向后移动焦点。可以扩展 `DefaultFocusManager`，而且可以重载其 `processKeyEvent` 方法以便修改缺省的行为。

假如当前有焦点的组件没有重载 `getNextFocusedComponent()` 来返回一个非空组件，则缺省焦点管理器在 `compareTabOrder` 方法的帮助下将决定哪个组件下一次将接受焦点。`DefaultFocusManager.compareTabOrder` 以两个组件为参数并返回一个 `boolean` 值，该值指出在 `tab` 顺序中它的第一个组件参数是否在第二个组件参数前。

图 4-27 所示的小应用程序实现一个带 `tab` 顺序的焦点管理器，这个焦点管理器的 `tab` 顺序与缺省的焦点管理器的 `tab` 顺序相反。该小应用程序的焦点管理器的 `tab` 顺序是从右到左、从下到上。

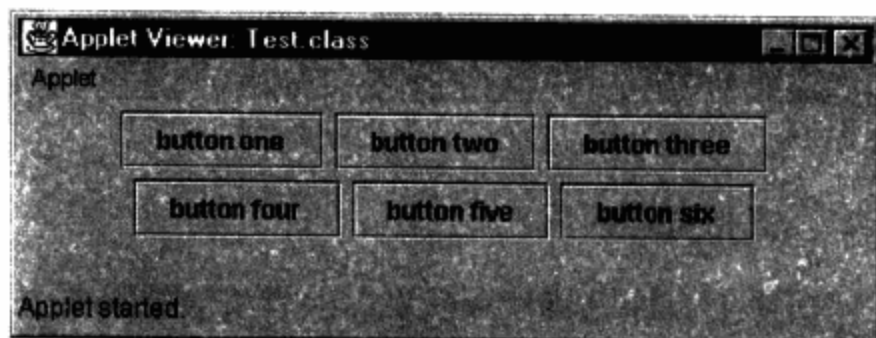


图 4-27 实现一个定制的焦点管理器

例 4-23 列出了图 4-27 所示的小应用程序的代码。

例 4-23 实现一个定制的焦点管理器

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private JButton button_1 = new JButton("button one"),
        button_2 = new JButton("button two"),
        button_3 = new JButton("button three"),
        button_4 = new JButton("button four"),
        button_5 = new JButton("button five"),
        button_6 = new JButton("button six");

    public void init() {
        Container contentPane = getContentPane();

        javax.swing.FocusManager.setCurrentManager (
            new CustomFocusManager());
        contentPane.setLayout (new FlowLayout());
        contentPane.add (button_1);
        contentPane.add (button_2);
        contentPane.add (button_3);
        contentPane.add (button_4);
        contentPane.add (button_5);
        contentPane.add (button_6);
    }
}

class CustomFocusManager extends DefaultFocusManager {
    public boolean compareTabOrder (Component a, Component b) {
```



```

Point location_a = a.getLocation();
Point location_b = b.getLocation();

int ax = location_a.x, ay = location_a.y;
int bx = location_b.x, by = location_b.y;

if (Math.abs(ay - by) < 10) {
    return (bx < ax);
}

return (ay > by);
}

```

该小应用程序调用 `FocusManager.setCurrentManager()` 来设置焦点管理器。该小应用程序使用完整的 `FocusManager` 类的名字，因为 AWT 也有同名的焦点管理器。

`CustomFocusManager` 类扩展 `DefaultFocusManager` 并重载 `compareTabOrder` 方法。每个组件的 `x` 和 `y` 坐标都是通过调用 `getLocation()` 而获得的，如果第二个组件在第一个组件的左边或上面，则返回一个 `true` 值。

4.11 支持可访问性

历史上，计算机技术的发展一直未能解决残疾用户的使用问题。例如，当窗口技术代替了文本显示后，又花费了许多年才出现能帮助盲人用户使用窗口技术的屏幕阅读器。而在 Swing 中，援助技术从开始就内置于组件中。

可访问性是使所有的用户都可以使用计算机的技术。这不仅包括有残疾的用户，还包括在移动系统中的用户。例如，虽然在汽车的导航系统中可能包含了可视的显示内容，但是，驾驶者必须在眼睛不能离开路面的情况下利用这些可视内容。这可以通过解释这些可视内容并发出用于导航的语音命令来实现。

Java 基础类在三个方面支持可访问性：

- 插入式界面样式。
- Java 可访问性 API。
- Java 可访问性实用工具。

Swing 类的插入式界面样式设计使开发人员为用户界面组件实现可选择的界面样式。例如，为方便视力不好的用户使用，可以不使用视觉性的界面样式而使用音频界面样式组件。另一方面，为方便听力不好的用户使用，带音频的用户界面组件可以在播放音频剪辑时使用显示标题的界面样式。插入式界面样式体系结构还包括对多重 UI 的支持，以便附加 UI 可以添加到一个组件中。例如，一个多重 UI 既可以驱动盲人使用的终端，也可以实现组件通常的可视化表现。有关 Swing 的多重界面样式的更多信息，请参见 7.3 节“附加 UI”。

Java 可访问 API 使用户界面组件能够为一种援助技术提供信息。`AccessibleContext` 类定义组件应该提供的一组基本信息。例如，一种援助技术应该能够识别用户界面组件，并且使用户能够辨认它们（可能是以声音的形式）。`AccessibleContext` 类提供了一个用来确认用户界面组件的角色的 `getAccessibleRole()` 方法，通常用它来确认组件的类型。`AccessibleContext.getAccessibleName()` 和 `AccessibleContext.getAccessibleDescription()` 提供与一个特定组件实例有关的更多信息。例如，一个滚动条的 `AccessibleContext` 可能确认该组件的角色是 `AccessibleRole.SCROLL_BAR`，而名字和描述将确认由特定的滚动条实例实现的功能。

`JComponent` 为获得可访问信息提供了内置的支持。所有的 Swing 组件利用 `JComponent` 的内

置功能来提供有关它们自身的可访问信息。例如，图 4-28 所示的小应用程序在“show accessible information”按钮激活时，将显示一个文本域的可访问角色。该小应用程序的输出如下：

Accessible Role: text

Accessible Description: Enter your first name

Accessible Name: First Name

例 4-24 列出了图 4-28 所示的小应用程序的代码。

例 4-24 获得可访问性信息

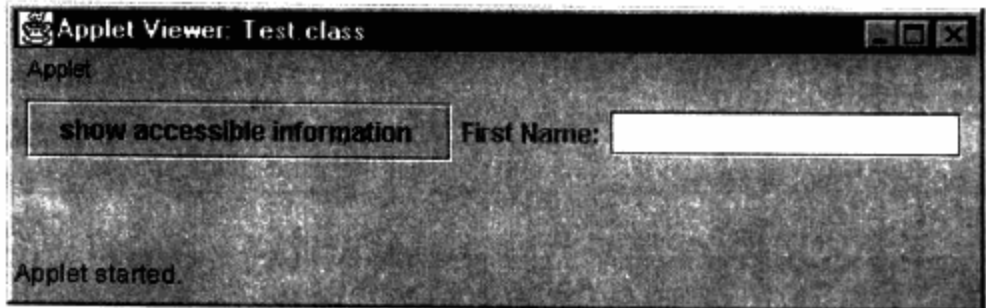


图 4-28 获得可访问信息

```
import javax.swing.* ;
import javax.accessibility.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane () ;
        JLabel label = new JLabel ("First Name:") ;
        JButton showButton = new JButton (
            "show accessible information") ;

        final JTextField field = new JTextField (15) ;
        AccessibleContext fieldContext =
            field.getAccessibleContext () ;
        fieldContext.setAccessibleName ("First Name") ;
        fieldContext.setAccessibleDescription (
            "Enter your first name") ;

        contentPane.setLayout (new FlowLayout ()) ;
        contentPane.add (showButton) ;
        contentPane.add (label) ;
        contentPane.add (field) ;

        showButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                AccessibleContext context ;
                AccessibleRole role ;

                context = field.getAccessibleContext () ;
                role = context.getAccessibleRole () ;

                System.out.print ("Accessible Role: ") ;
                System.out.println (
                    context.getAccessibleRole ()) ;

                System.out.print ("Accessible Description: ") ;
                System.out.println (
                    context.getAccessibleDescription ()) ;

                System.out.print ("Accessible Name: ") ;
                System.out.println (
```

```
context.getAccessibleName());
```

```
});
```

该小应用程序创建一个文本域并设置这个域的可访问名字和描述。一个动作监听器添加到“show accessible information”按钮中，以便获得这个域的可访问的相关内容。用可访问相关内容可以获得这个域的可访问角色、描述和名字。

4.12 本章回顾

JComponent 类是所有 Swing 轻量组件的基类。JComponent 类扩展 java.awt.Container 类，java.awt.Container 类又扩展 java.awt.Component。结果，每个 Swing 轻量组件都具有这些类的功能。

除了继承了 AWT 超类的功能外，JComponent 类本身还提供了大量可靠的下层构件。100 多个 JComponent 方法提供对不透明和透明组件、双缓存、调试图形、自动滚动、工具提示等的支持。

JComponent 类还修补了一些从其 AWT 超类继承来的设计缺陷。例如，AWT 的 Component 类有一个重要的面向对象错误，即它要求使用继承来修改其实例的最小、最大和首选尺寸。通常，不能要求用继承来修改在一个类的多个实例中广泛变化的特性，因为它可能产生许许多多的子类。JComponent 类提供了在 Component 类中缺乏的设置方法，因此，Swing 轻量组件的大小不需使用继承就能设置。

虽然 Swing 组件建立在 AWT 下层构件之上，但是，必须记住，所有的 JComponent 扩展实际上都是 AWT 容器。而且，Swing 轻量组件可以有一个边框和一个 UI 代表。例如，在重载 Swing 轻量组件的绘制行为时，这个特性就很重要。与 AWT 组件要重载 paint 方法不同，通常为 JComponent 扩展重载 paintComponent 方法。另外，在计算最小、最大和首选尺寸时，必须考虑 Swing 轻量组件的边衬问题。

JComponent 类建立在 AWT 超类提供的功能之上，同时修补了一些继承来的面向对象的编程方面的漏洞。结果，JComponent 提供了一个可靠的基础，所有的 Swing 轻量组件都建立在此基础之上。

第5章 边框、图标和动作

本章介绍 Swing 的三种实用工具：边框、图标和动作。

边框绘制在组件的边界周围，它有许多不同的种类：线边框、雕刻边框、不光滑的边框等等。边框本身不是组件，所以，它们绘制在指定组件的边衬中。

图标是图形对象，通常是一个小图像。与边框一样，图标在指定组件的指定位置上绘制。

动作封装图形用户界面的一个逻辑操作，并且还简化用户界面元素的构造工作。动作通常由一个或多个图标或文本字符串组成。可以把动作添加到某些容器中，这些容器创建一个组件与这个动作相关联。例如，利用 `JMenu.add (Action)` 方法，可把动作添加到一个菜单中。当一个动作添加到一个菜单中时，这个菜单用与这个动作相关联的文本和图标来创建一个菜单项并把这个菜单项添加到菜单中。

边框、图标和动作都是很有意义的，因为它们都可以与多个组件相关联。由于边框和图标都不是组件，但却都能绘制到组件中，所以，可以在支持使用边框和图标的多个组件中共享边框和图标。动作也必须被多个组件所共享，并且用来作为控制的中心点以维护与这个动作相关联的组件的启用状态。

5.1 边框

通过构造所需类型的边框，然后把这个边框传送给 `JComponent.setBorder (Border)`，所有 `JComponent` 扩展 (`JViewport` 除外) 都可以有边框。虽然每个组件可以只有一个边框，但 Swing 支持组合边框，因此，在实际应用中，单个组件可以使数个边框嵌套在一起，使边框有一定的深度。

边框的使用很简单。例如，图 5-1 示出了一个带标题边框的 `JPanel` 实例。

例 5-1 列出了图 5-1 所示的小应用程序的代码。

例 5-1 一个带边框的 `JPanel` 的小应用程序

```
import java.awt.BorderLayout;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet {
    public void init () {
        JPanel panel = new JPanel ();
        panel.setBorder (new TitledBorder ("JPanel Border"));
        getContentPane ().add (panel, BorderLayout.CENTER);
    }
}
```

这个小应用程序创建一个带标题的边框，这个边框传送给面板的 `setBorder` 方法。

5.1.1 边框和边衬

AWT 容器有一个 `insets` 属性，它定义容器的边衬。布局管理器仔细地布局一个容器中的各

个组件，以便这些组件不会侵占这个容器的边衬区。容器的 insets 属性是一个只读属性，修改 AWT 容器 insets 属性唯一的方法是子类化一个容器并重载它的 getInsets 方法。

除 JViewport（它不允许自己有一个边框）外，Swing 组件不需要重载 getInsets（）来定义一个边衬区。相反，如果在组件的边缘有空着的空间，则 Swing 组件使用一个空边框。当请求 Swing 组件的边衬时，Swing 组件返回它们边框的边衬。如果一个 Swing 组件确实没有边框，只有这时，Swing 组件才会返回由其超类 java.awt.Container 维护的 insets 属性。

用 Swing 建立的定制组件应该沿用而不是重载 getInsets（）。如果要填充定制组件的边界周围，则定制组件应该使用一个空边框。

使用空边框来指定边衬，可以有效地在每个实例的基础上设置 insets 属性。AWT 容器只能通过子类化和重载 getInsets（）来修改它们的边衬，而 Swing 组件可以通过把其边框设置为一个 EmptyBorder 实例来修改它们的边衬。

5.1.2 Swing 的边框类型

Swing 提供许多种边框类型，如表 5-1 所示。

表 5-1 Swing 的边框类型

边框类型	描述	不透明	相关的常量
雕刻	一个 3D 边框	是	RAISED、LOWERED
组合	含有一个内部边框和一个外部边框	变化的	n/a
空	一个不用绘制的透明边框	不是	n/a
蚀刻	一个蚀刻的边框	是	RAISED、LOWERED
线	可设置宽度的有颜色的单线边框	是	n/a
不光滑的	一种边框，它在边框的空白处填充一种颜色或在边框的空白处平铺一幅图像	是	n/a
软雕刻	有软角的雕刻边框	不是	RAISED、LOWERED
带标题	带标题的边框。标题的位置是可设置的。	不是	DEFAULT_POSITION、ABOVE_TOP、TOP、BELOW_TOP、ABOVE_BOTTOM、BOTTOM、BELOW_BOTTOM、DEFAULT_JUSTIFICATION、LEFT、CENTER、RIGHT

表 5-1 所列的每种边框类型都对应 swing.border 包中的一个类。雕刻边框、软雕刻边框和蚀刻边框可以指定为是突出的还是凹进的，带标题边框的标题可以放在边框内的各种不同的位置上。雕刻边框和软雕刻边框之间的差别一般人不易察觉，但是软雕刻边框使雕刻边框的四角是圆弧状的。

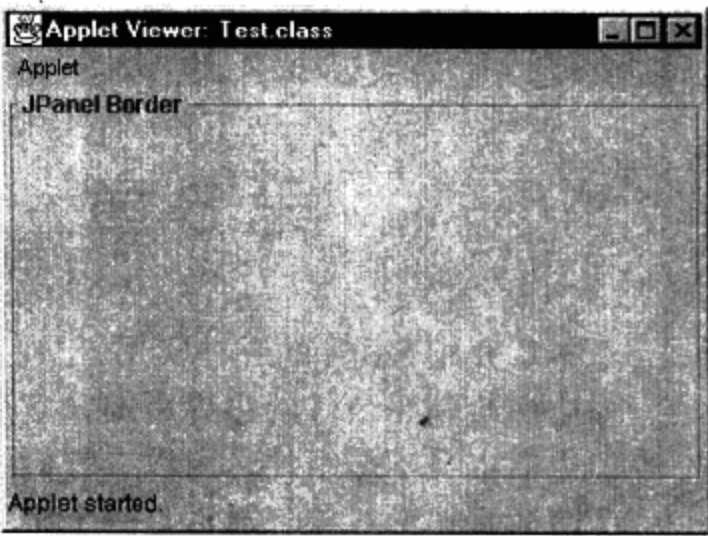


图 5-1 一个带标题边框的 JPanel 实例

不透明边框绘制它们边衬区中的每一个像素点。如果组合边框的内外边框都是不透明的，则组合边框是不透明的，所以，表 5-1 中组合边框的不透明属性是变化的。

奇怪的是，不透明边框的概念并没有在 Swing 的其他部分贯彻。换句话说，在 Swing 中没有任何代码根据边框是透明的还是不透明的来改变其行为。事实上，如果你研究过 Swing 源代码，那么你会发现 `Border.isBorderOpaque` 方法从来没有被调用过。

不透明边框的概念是一种初始边框设计的残余，这种设计是没有远见的。尽管如此，如果你从未根据边框是否是不透明的来优化边框的绘制，则你已经开始使用 `Border.isBorderOpaque` 方法了。

图 5-2 示出了一个显示各种类型 Swing 边框的小应用程序。

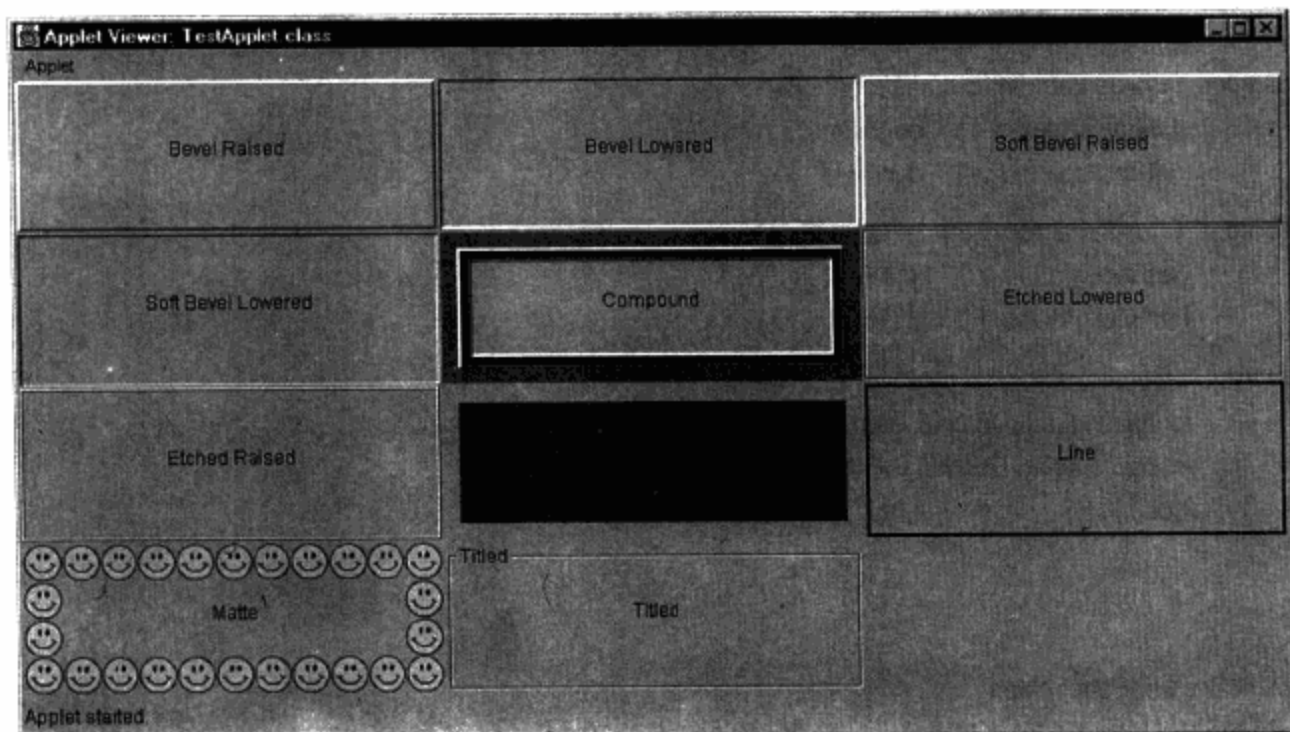


图 5-2 Swing 边框类型

图 5-2 所示的小应用程序只包含一个 `JPanel` 扩展——`AllBordersPanel`，而 `AllBordersPanel` 又包含 11 个面板。这 11 个面板中的每个面板都是一个 `PanelWithTitle` 实例，`PanelWithTitle` 是另一个 `JPanel` 扩展，重载 `JPanel` 扩展的 `paint` 方法绘制一个居中的标题。

实例化这 11 个面板并为每一个面板配备一个边框的操作如下：

```
class AllBordersPanel extends JPanel {
    public AllBordersPanel () {
        JPanel bl = new PanelWithTitle ("Bevel Lowered"),
        br = new PanelWithTitle ("Bevel Raised"),
        c = new PanelWithTitle ("Compound"),
        l = new PanelWithTitle ("Line"),
        m = new PanelWithTitle ("Matte"),
        e = new PanelWithTitle ("Empty"),
        t = new PanelWithTitle ("Titled"),
        sbr = new PanelWithTitle ("Soft Bevel Raised"),
        sbl = new PanelWithTitle ("Soft Bevel Lowered"),
        el = new PanelWithTitle ("Etched Lowered"),
        er = new PanelWithTitle ("Etched Raised");

        ...
        bl.setBorder (BorderFactory.createLoweredBevelBorder ());
        br.setBorder (BorderFactory.createRaisedBevelBorder ());
        sbr.setBorder (new SoftBevelBorder (BevelBorder.RAISED));
        sbl.setBorder (new SoftBevelBorder (BevelBorder.LOWERED));
    }
}
```



```

t.setBorder (BorderFactory.createTitledBorder ("Titled"));
l.setBorder (
    BorderFactory.createLineBorder (Color.black, 2));
c.setBorder (
    BorderFactory.createCompoundBorder (
        //outer border
        BorderFactory.createCompoundBorder (
            BorderFactory.createLineBorder (Color.gray, 10),
            BorderFactory.createRaisedBevelBorder ()),
        //inner border
        BorderFactory.createCompoundBorder (
            BorderFactory.createLineBorder (Color.blue, 5),
            BorderFactory.createLoweredBevelBorder ()))));
el.setBorder (BorderFactory.createEtchedBorder (
    getBackground ().brighter (),
    getBackground ().darker ());
er.setBorder (BorderFactory.createEtchedBorder (
    getBackground ().darker (),
    getBackground ().brighter ());
m.setBorder (BorderFactory.createMatteBorder (
    iconsz.height, iconsz.width,
    iconsz.height, iconsz.width,
    icon));
...
|
...
| //end of AllBordersPanel class

```

这些边框都是从边框库获得的，而不是直接实例化的（软雕刻边框除外）。Swing 1.1 版本中的 `BorderFactory` 类不包括创建软雕刻边框的方法。在第 5.1.7 节“边框库——共享边框”中会介绍边框库。

边框库的另一个麻烦是它的方法不对称。例如，`BorderFactory` 类有分别创建突出的和凹进的雕刻边框的方法——`createRaisedBevelBorder` 和 `createLoweredBevelBorder`。但对蚀刻边框却没有类似的方法。要显式地创建突出的或凹进的蚀刻边框，必须使用 `createEtchedBorder` 方法，这个方法要用两种颜色，一种颜色用于增亮，一种颜色用于阴影^①。

通过指定要在边框的边衬区中平铺的 `ImageIcon` 来创建不平滑边框。为了增亮边框，具有空边框的面板用一个实颜色来填充面板的内部。

组合边框包含两个边框，这两个边框本身都是组合边框。因此，具有组合边框的面板实际上有四个嵌套的边框。传送给 `CompoundBorder` 构造方法的第一个边框是外部边框，第二个边框代表内部边框。

例 5-2 列出了这个小应用程序的完整代码。

例 5-2 显示所有 Swing 边框类型的小应用程序

```

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

```

① 1. `BorderFactory.createEtchedBorder ()` 方法创建一个突出的蚀刻边框。

```

public class Test extends JApplet {
    public void init () {
        JPanel jpanel = new AllBordersPanel ();
        getContentPane ().add (jpanel, BorderLayout.CENTER);
    }
}

class AllBordersPanel extends JPanel {
    public AllBordersPanel () {
        JPanel bl = new PanelWithTitle ("Bevel Lowered"),
        br = new PanelWithTitle ("Bevel Raised"),
        c = new PanelWithTitle ("Compound"),
        l = new PanelWithTitle ("Line"),
        m = new PanelWithTitle ("Matte"),
        e = new PanelWithTitleEmptyBorder ("Empty"),
        t = new PanelWithTitle ("Titled"),
        sbr = new PanelWithTitle ("Soft Bevel Raised"),
        sbl = new PanelWithTitle ("Soft Bevel Lowered"),
        el = new PanelWithTitle ("Etched Lowered"),
        er = new PanelWithTitle ("Etched Raised");

        setLayout (new GridLayout (4, 3, 2, 2));
        ImageIcon icon = new ImageIcon ("smiley.gif");
        Dimension iconsz = new Dimension (icon.getIconWidth (),
                                            icon.getIconHeight ());

        bl.setBorder (BorderFactory.createLoweredBevelBorder ());
        br.setBorder (BorderFactory.createRaisedBevelBorder ());
        sbr.setBorder (new SoftBevelBorder (BevelBorder.RAISED));
        sbl.setBorder (new SoftBevelBorder (BevelBorder.LOWERED));
        t.setBorder (BorderFactory.createTitledBorder ("Titled"));
        l.setBorder (
            BorderFactory.createLineBorder (Color.black, 2));
        c.setBorder (
            BorderFactory.createCompoundBorder (
                BorderFactory.createCompoundBorder (
                    BorderFactory.createLineBorder (Color.gray, 10),
                    BorderFactory.createRaisedBevelBorder ()),
                BorderFactory.createCompoundBorder (
                    BorderFactory.createLineBorder (Color.blue, 5),
                    BorderFactory.createLoweredBevelBorder ()))));
        el.setBorder (BorderFactory.createEtchedBorder (
            getBackground ().brighter (),
            getBackground ().darker ());
        er.setBorder (BorderFactory.createEtchedBorder (
            getBackground ().darker (),
            getBackground ().brighter ());
        m.setBorder (BorderFactory.createMatteBorder (
            iconsz.height, iconsz.width,
            iconsz.height, iconsz.width,
            icon));

        add (br); add (bl); add (sbr);
        add (sbl); add (c); add (el);
        add (er); add (e); add (l);
    }
}

```

```
        add (m); add (t);
    }
}

class PanelWithTitle extends JPanel {
    private String title;
    public PanelWithTitle (String title) {
        this.title = title;
    }
    public void paintComponent (Graphics g) {
        FontMetrics fm = g.getFontMetrics ();
        Dimension size = getSize ();
        int titleW = fm.stringWidth (title);

        g.setColor (Color.black);
        g.drawString (title, size.width/2 - titleW/2,
                      size.height/2);
    }
}

class PanelWithEmptyBorder extends PanelWithTitle {
    public PanelWithEmptyBorder (String title) {
        super (title);
        setBorder (BorderFactory.createEmptyBorder (10, 10, 10, 10));
    }
    public void paintComponent (Graphics g) {
        Dimension size = getSize ();
        Insets insets = getInsets ();
        g.setColor (Color.red);
        g.fillRect (insets.left, insets.top,
                    size.width - 2 * insets.left,
                    size.height - 2 * insets.top);
        super.paintComponent (g);
    }
}
```

Swing 提示

边框不是组件

Swing 的边框类在组件中绘制，但边框本身不是组件。要绘制边框的组件传送给 `Border.paintBorder` 方法。因此，许多组件可以共享单个边框；当每个组件都要绘制边框时，组件把自己传送给 `Border.paintBorder ()` 方法作为要绘制边框的组件。

5.1.3 不透明与透明之间的比较

与 Swing 轻量组件一样，边框设计成不透明的或部分透明的。只有当边框绘制每一个像素点时，这个边框才是不透明的；如果边框未能绘制每个像素点甚至就是没有绘制一个像素点，则这个边框就不是不透明的。

图 5-2 示出了包含一个 `JPanel` 扩展（它配备了一个标题边框）的小应用程序。

这个面板在其背景上平铺一幅图像。例 5-3 列出了图 5-3 所示的小应用程序的代码。

例 5-3 部分透明的边框的样例

```
import javax.swing.* ;
import javax.swing.border.* ;
import java.awt.* ;

public class Test extends JApplet {
    JPanel panel = new JPanel () ;
    TitledBorder border = new TitledBorder ("JPanel Border");

    public void init () {
        panel.setBorder (border);
        getContentPane ().add (panel, BorderLayout.CENTER);

        System.out.println ("opaque = " + border.isBorderOpaque ());
        System.out.println (
            "insets = " + border.getBorderInsets (panel));
    }
}

class RainPanel extends JPanel {
    public void paintComponent (Graphics g) {
        Icon icon = new ImageIcon ("rain.gif");
        Dimension size = getSize ();

        int patchW = icon.getIconWidth (),
            patchH = icon.getIconHeight ();

        for (int r=0; r < size.width; r += patchW) {
            for (int c=0; c < size.height; c += patchH)
                icon.paintIcon (this, g, r, c);
        }
    }
}
```

这个小应用程序显示这个带标题的边框是否是不透明的，并且列出了边框占据的边衬：

```
opaque = false
insets = java.awt.Insets [top = 21, left = 6, bottom = 6, right = 6]
```

虽然这个边框占用了上述的边衬，但是，面板的背景还是可以透过边框显示出来，这是因为这个边框不是完全不透明的。

注意，RainPanel 正在做一些通常认为是坏设计的事情——绘制到它的边衬中。因为由 JComponent 派生的组件是容器，所以使用这些组件应该十分小心，不要把这些组件绘制到它们的边衬中（参见 4.4.3 节“在 Swing 组件中重载绘制方法”）。在这个例子中，绘制到边衬中产生了一个所需的效果，它图解说明了半透明的边框。

5.1.4 边框包

与 Swing 边框有关的所有类（BorderFactory 类除外）都在 swing.border 包中。这个包包括一个 Border 接口、一个 AbstractBorder 类和许多与表 5-1 所列的边框类型相对应的具体的边框类。

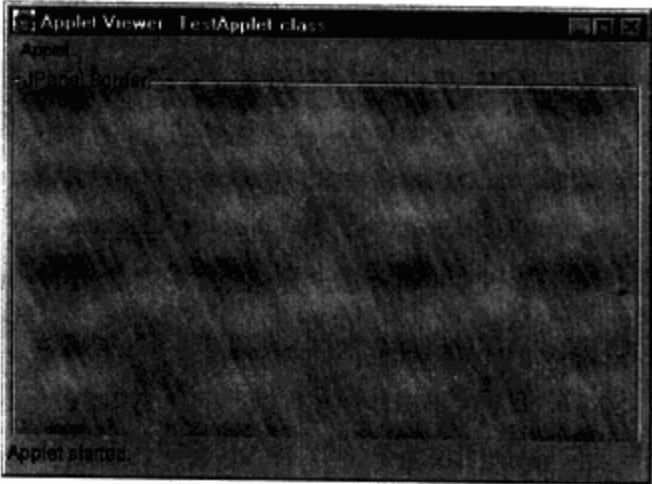


图 5-3 一个带标题边框的 JPanel 实例

图 5-4 示出了边框包的类图。

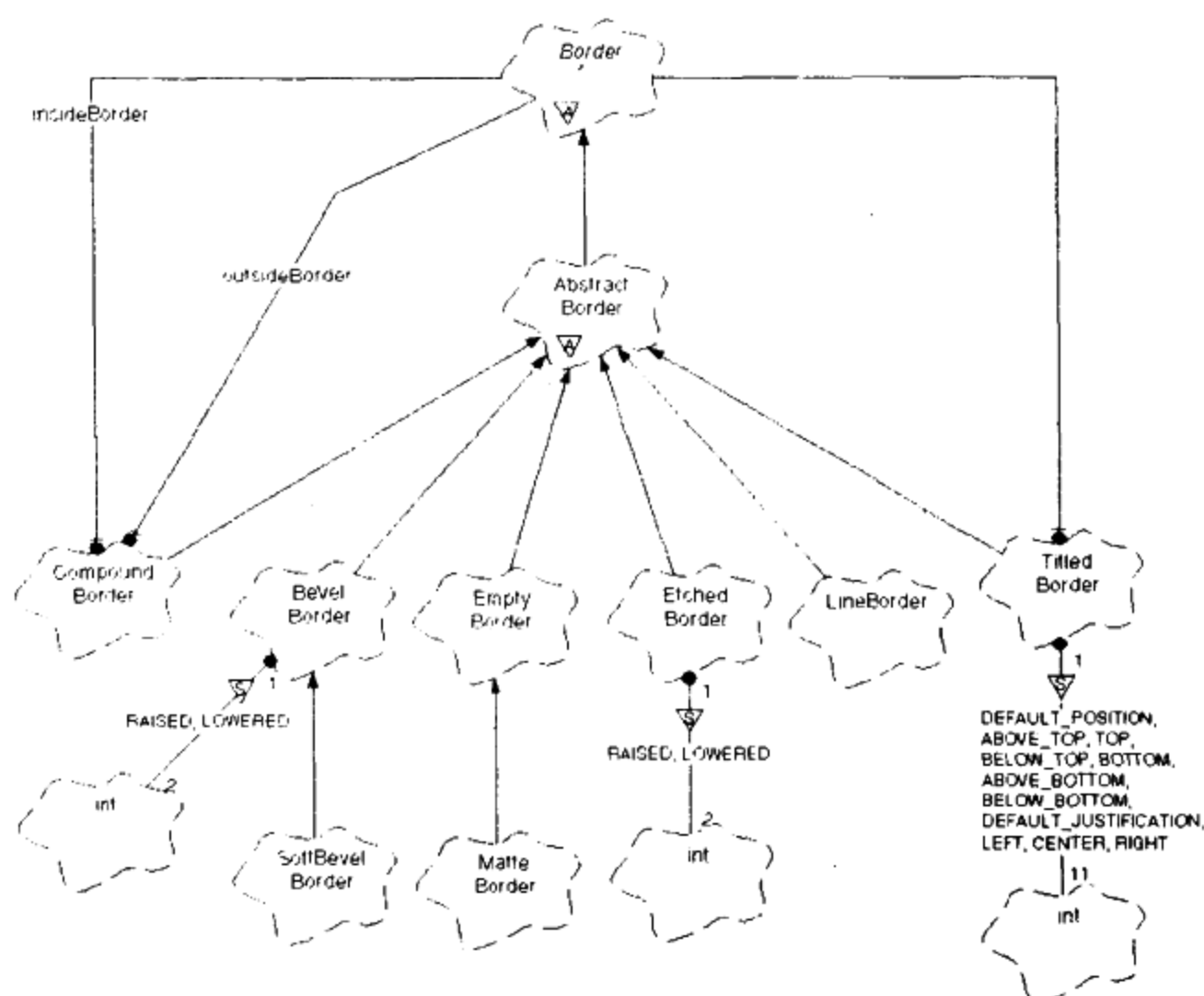


图 5-4 Swing 边框包概览

5.1.5 边框接口

所有的边框都实现 Border 接口，接口总结 5-1 列出了这个接口的方法。

接口总结 5-1 swing.border.Border

```

public abstract void paintBorder (Component, Graphics, int x, int y, int w, int h)
public abstract Insets getBorderInsets (Component)
public abstract boolean isBorderOpaque ()

```

通过调用 `paintBorder (Component, Graphics, int, int, int, int)` 来绘制边框。注意，在 `Border` 中 `paintBorder` 与 `JComponent.paint ()` 的名字是不同的，因为要强调这样一个事实：即边框不是组件。`PaintBorder` 的四个 `integer` 参数指定了边框的位置、宽度和高度。

`paintBorder ()` 中的参数 `Graphics` 通常是与这个组件相关联的 `Graphics` 或与一个屏外缓冲（如果这个组件是双缓冲的话）相关联的 `Graphics`。`Component` 参数在需要时可以用来提取有关这个组件的信息，例如组件的背景色和前景色。`integer` 参数指定这个边框左上角的位置和这个边框的宽度和高度。

5.1.6 AbstractBorder 类

所有的 Swing 边框都扩展实现 Border 接口的 `AbstractBorder` 类。`AbstractBorder` 实现绘制边框、返回边衬和指出边框是否是不透明的等缺省功能。类总结 5-1 列出了 `AbstractBorder` 类方法。

类总结 5-1 swing.border.AbstractBorder

1. 构造方法

```
public AbstractBorder ()
```

AbstractBorder 无参数构造方法是编译器产生的。

2. 方法

```
public static Rectangle getInteriorRectangle (Component, Border, int x, int y, int w, int h)
```

```
public Rectangle getInteriorRectangle (Component, Border, int x, int y, int w, int h)
```

```
public Insets getBorderInsets (Component)
```

```
public Insets getBorderInsets (Component, Insets)
```

```
public boolean isBorderOpaque ()
```

```
public void paintBorder (Component, Border, int x, int y, int w, int h)
```

AbstractBorder 提供一个方便的 static 方法来确定边框的内部矩形。还提供了一个调用这个 static 方法的一个实例的方法。

上面所列的第一个 getBorderInsets 方法返回(0,0,0,0)的边衬, 第二个 getBorderInsets 方法把传送给它的边衬值设置为(0,0,0,0)。两个方法都必须由 AbstractBorder 的扩展重载。

AbstractBorder 实现 isBorderOpaque 方法以便返回 false, paintBorder 方法还有一个必须被 AbstractBorder 的扩展所重载的空实现。

5.1.7 边框库——共享边框

因为边框以要绘制边框的组件为参数, 所以单个边框可以用于装饰多个组件。事实上, Swing 包提供了一个 BorderFactory 类, 它构造共享边框实例。例如, 例 5-4 所列的小应用程序从边框库中获得两个突出的雕刻边框。

例 5-4 从边框库中获得边框

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JPanel panel = new JPanel ();
        JPanel panel2 = new JPanel ();
        Border border = BorderFactory.createRaisedBevelBorder ();
        Border border2 = BorderFactory.createRaisedBevelBorder ();

        panel.setBorder (border);
        panel2.setBorder (border2);

        contentPane.add (panel, BorderLayout.NORTH);
        contentPane.add (panel2, BorderLayout.SOUTH);

        if (border == border2)
            System.out.println ("bevel borders are shared");
        else
```



```
System.out.println("bevel borders are NOT shared");
```

从边框库中获得的这两个边框其实是一个边框而且完全相同，正如这个小应用程序的输出所验证的那样：

```
bevel borders are shared
```

BorderFactory 类提供了一些 static 方法，这些方法返回所有 Swing 边框类型的实例（软雕刻边框除外），这些实例可能有不同的配置。对 Swing 1.1 FCS 版本而言，由边框库返回雕刻边框和蚀刻边框的实例是共享实例，对带标题的边框和 matte 边框等其他的边框类型，边框库返回的是不能共享的实例。

边框库不返回所有边框类型的共享实例这个事实提出了一个问题：为什么非共享边框也使用边框库？为什么边框库产生的所有边框不是共享实例呢？与这个问题一样，答案也是双重的。

首先，使用边框库不需要引入 Swing 的 border 包，只要不维护对返回的边框的引用。因此，不论边框库是否返回共享的边框实例，都可以避免不得不引入 Swing.border。

其次，返回蚀刻边框和雕刻边框等简单的边框的共享实例是很容易实现的。但具有标题等属性的边框使返回共享实例很困难。由于时间限制，在 Swing 1.1 FCS 版本中，Swing 小组选择只返回简单边框类型的共享实例。

未来的 Swing 版本可以返回其他类型的边框的共享实例，因此，应尽量使用边框库。

Swing 提示

使用边框库

在下次寻找装饰 Swing 组件的边框时，请记住边框库。使用边框库不用引入 swing.border 包，而且还更有效率，因为边框库返回某些边框类型的共享实例。

5.1.8 替换内置边框

缺省时，有些 Swing 组件配备了内置边框，这些内置边框根据组件的状态进行不同的绘制。这些组件的替代边框可能对组件的界面样式有不好的效果。例如，图 5-5 所示的小应用程序包含两个按钮。上边的按钮是定制边框，下边的按钮是标准的 Windows 界面样式边框。

图 5-5 示出了两个处于按下状态的按钮。从图中很容易看出“regular button”按钮按下了，但很难看出“button with custom border”按钮也按下了。显式地给上边的按钮配备一个边框彻底破坏了这个按钮的界面样式。因此，当替换内置边框时要十分小心。

因为内置边框在不同的界面样式中是不同的，所以不存在一个包含内置边框的组件的完全列表。如果为一个组件指定一个边框，最好在各种不同的界面样式中测试这个边框以确保这个边框不会破坏这个组件的界面样式特征。

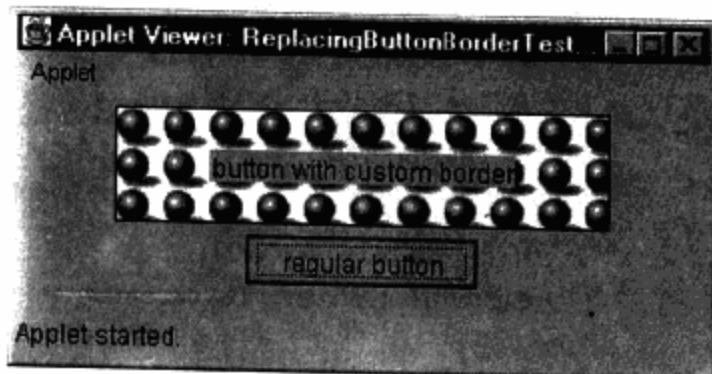


图 5-5 替换内置边框

5.1.9 实现定制边框

虽然 Swing 边框代表最常用的边框类型，但是你可能更喜欢定制边框。例如，绘制程序经常在对象周围绘制图柄，可用这些手柄来移动所选的对象或重新调整所选对象的大小。

图柄边框是 Swing 定制边框完美的代表。与缺省的 Swing 边框一样，定制边框应该扩展 `AbstractBorder` 类并有选择地重载它的方法。通常，这意味着重载 `paintBorder` 和 `getBorderInsets`，因为 `AbstractBorder` 不绘制并且返回一个 `(0,0,0,0)` 边衬。另外，如果定制边框要绘制边框中的每一个像素点的话，则这个定制边框应该重载 `isOpaque()` 方法以返回 `true`。缺省时，`AbstractBorder` 不是不透明的。图 5-6 示出了一个显示三个 `HandleBorder` 定制边框类实例的小应用程序。

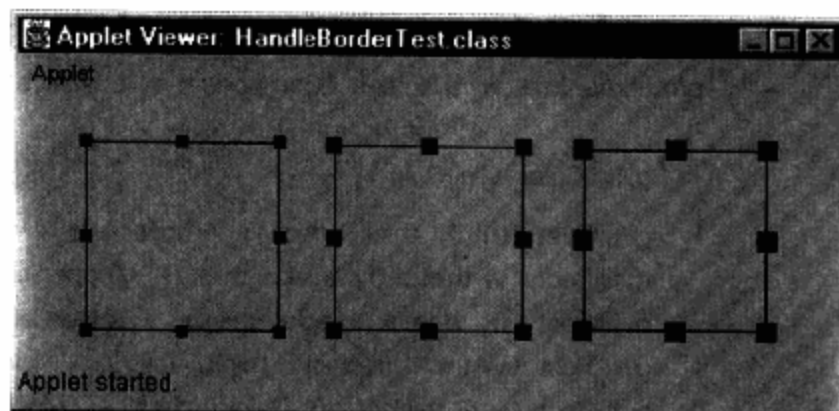


图 5-6 一个定制边框

图 5-6 所示的图柄边框不是不透明的，因此，`HandleBorder` 类没有重载 `isOpaque()`；然而，它重载了 `paintBorder()` 和 `getBorderInsets()`。

例 5-5 列出了 `HandleBorder` 类。

例 5-5 `HandleBorder` 类清单

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class HandleBorder extends AbstractBorder {
    Protected Color lineColor;
    protected int thick;

    public HandleBorder () {
        this (Color.black, 6);
    }

    public HandleBorder (Color lineColor, int thick) {
        this.lineColor = lineColor;
        this.thick = thick;
    }

    public void paintBorder (Component c, Graphics g, int x,
        int y, int w, int h) {
        Graphics copy = g.create ();
        if (copy != null) {
            try {
                copy.translate (x, y);
                paintRectangle (c, copy, w, h);
                paintHandles (c, copy, w, h);
            }
            finally {
                copy.dispose ();
            }
        }
    }

    public Insets getBorderInsets () {
```

```

    return new Insets (thick, thick, thick, thick);
}

protected void paintRectangle (Component c, Graphics g,
                                int w, int h) {
    g.setColor (lineColor);
    g.drawRect (thick/2, thick/2, w-thick-1, h-thick-1);
}

protected void paintHandles (Component c, Graphics g,
                              int w, int h) {
    g.setColor (lineColor);

    g.fillRect (0, 0, thick, thick); //upper-left
    g.fillRect (w-thick, 0, thick, thick); //upper-right
    g.fillRect (0, h-thick, thick, thick); //lower-left
    g.fillRect (w-thick, h-thick, thick, thick); //lower-right
    g.fillRect (w/2-thick/2, 0, thick, thick); //mid-top
    g.fillRect (0, h/2-thick/2, thick, thick); //mid-left
    g.fillRect (w/2-thick/2, h-thick, thick, thick); //mid-bottom
    g.fillRect (w-thick, h/2-thick/2, thick, thick); //mid-right
}
}

```

HandleBorder 除了有它自己的用途外，还设计成一个基类。PaintBorder 方法调用 protected paintRectangle 和 paintHandles 方法，这两个方法可以被重载以适应扩展的需要。

因为边框以用于绘制组件的实际的 Graphics 为参数，所以 Graphics 的一个拷贝用于绘制图柄边框。为方便 paintRectangle 和 paintHandles 方法，转换了 Graphics 的拷贝。接着，处理 Graphics 拷贝，就像所有的 Graphics（它由 AWT 提供的数个 getGraphics 方法中的一种方法来获取）所要求的那样。

定制边框的使用方式与 Swing 提供的边框的使用方式完全相同。例 5-6 列出了图 5-6 所示的小应用程序的代码。

例 5-6 使用定制边框

```

import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();

        JPanel [] panels = { new JPanel (),
                              new JPanel (), new JPanel () };

        Border [] borders = { new HandleBorder (),
                              new HandleBorder (Color.red, 8),
                              new HandleBorder (Color.blue, 10) };

        contentPane.setLayout (
            new FlowLayout (FlowLayout.CENTER, 20, 20));

        for (int i = 0; i < panels.length; ++i) {
            panels [i].setPreferredSize (new Dimension (100, 100));
            panels [i].setBorder (borders [i]);
            contentPane.add (panels [i]);
        }
    }
}

```

5.2 图标

在概念上，图标与边框非常相似，图标和边框都绘制到通常与一个组件相关联的 Graphics 中。接口总结 5-2 列出了由 Icon 接口定义的方法。

接口总结 5-2 Icon

```
public abstract int getIconHeight ()
public abstract int getIconWidth ()
public abstract void paintIcon (Component, Graphics, int, int)
```

除了把图标自己绘制到一个组件中以外，图标还可以报告它的宽度和高度。

Swing 只提供了 Icon 接口的一个实现，即绘制图像的 ImageIcon 类。然而，图标不一定要具有图像。例如，图 5-7 示出了一个小应用程序，它包括三个 ColorIcon 实例，ColorIcon 是实现 Icon 接口的类。

例 5-7 列出了图 5-7 所示的小应用程序的代码。这三个 ColorIcons 在这个小应用程序的 paint 方法中绘制。

例 5-7 一个绘制图标的小应用程序

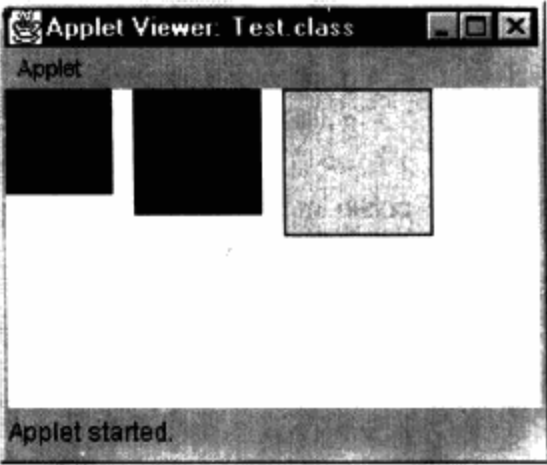


图 5-7 一个 Icon 扩展的三个实例

```
import java.awt.*;
import javax.swing.*;

public class IconTest extends JApplet {
    ColorIcon redIcon = new ColorIcon (Color.red, 50, 50),
              blueIcon = new ColorIcon (Color.blue, 60, 60),
              yellowIcon = new ColorIcon (Color.yellow, 70, 70);

    public void paint (Graphics g) {
        redIcon.paintIcon (this, g, 0, 0);
        blueIcon.paintIcon (this, g,
                             redIcon.getIconWidth () + 10, 0);
        yellowIcon.paintIcon (this, g,
                              redIcon.getIconWidth () + 10 +
                              blueIcon.getIconWidth () + 10, 0);
    }
}
```

例 5-8 列出了 ColorIcon 类。ColorIcon 是一个简单的类，但是，它能说明没有图像的图标。

例 5-8 ColorIcon 类清单

```
import java.awt.*;
import javax.swing.*;

class ColorIcon implements Icon {
```

```

private Color fillColor;
private int w, h;

public ColorIcon (Color fillColor, int w, int h) {
    this.fillColor = fillColor;
    this.w = w;
    this.h = h;
}

public void paintIcon (Component c, Graphics g, int x, int y) {
    //draw black border...

    g.setColor (Color.black);
    g.drawRect (x, y, w-1, h-1);

    //fill icon...

    g.setColor (fillColor);
    g.fillRect (x+1, y+1, w-2, h-2);
}

public int getIconWidth () {
    return w;
}

public int getIconHeight () {
    return h;
}

```

用填充色、宽度和高度来实例化一个 `ColorIcon`。它的 `paintIcon` 方法在图标边缘的周围简单地绘制一个黑色矩形，并用在构造时指定的填充色来填充图标的内部。

5.2.1 把图标与组件相关联

实际上，例 5-7 仅仅是在小应用程序中绘制图标。图标真正的价值是要把它们与组件相关联，图 5-8 显示了一个带菜单条的小应用程序。菜单条上唯一的菜单包含三个配备了 `ColorIcon` 实例的菜单项。

例 5-9 列出了图 5-8 所示的小应用程序的代码。

例 5-9 菜单项中的图标

```

import java.awt.*;
import javax.swing.*;

public class Test extends JApplet {
    ColorIcon redIcon = new ColorIcon (Color.red, 40, 15),
        blueIcon = new ColorIcon (Color.blue, 40, 15),
        yellowIcon = new ColorIcon (Color.yellow, 40, 15);

    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu colors = new JMenu ("Colors");
        colors.add (new JMenuItem (redIcon));
        colors.add (new JMenuItem (blueIcon));
        colors.add (new JMenuItem (yellowIcon));
    }
}

```

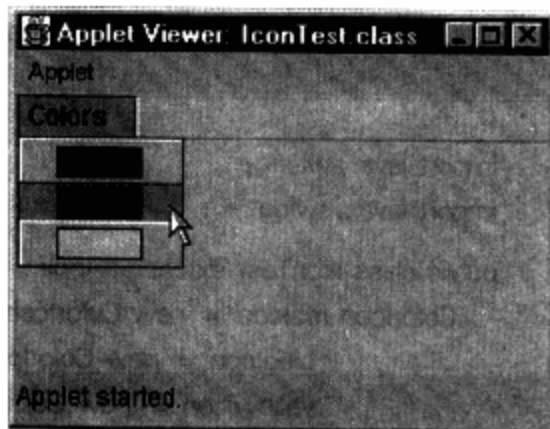


图 5-8 在菜单项中显示的图标

```
mb.add (colors);  
setJMenuBar (mb);
```

菜单项用一个图标来构造。当绘制菜单项时，每个菜单项的图标绘制到菜单项所占据的区域中。许多 Swing 组件（如下所列）都可以显示图标。

- JButton
- JCheckBox
- JCheckBoxMenuItem
- JFileChooser
- JLabel
- JList
- JMenuItem
- JOptionPane
- JPopupMenu
- JRadioButton
- JTable
- JToolBar

5.2.2 在组件中共享图标

许多支持显示图标的组件可以共享图标。例如，图 5-9 所示的小应用程序有一个带图标（该图标代表一种颜色）的按钮。当按钮激活时，则显示一个用于选择颜色的弹出式菜单。当从弹出式菜单中选取了一个颜色后，则按钮上显示的图标更新以反映所选取的颜色。

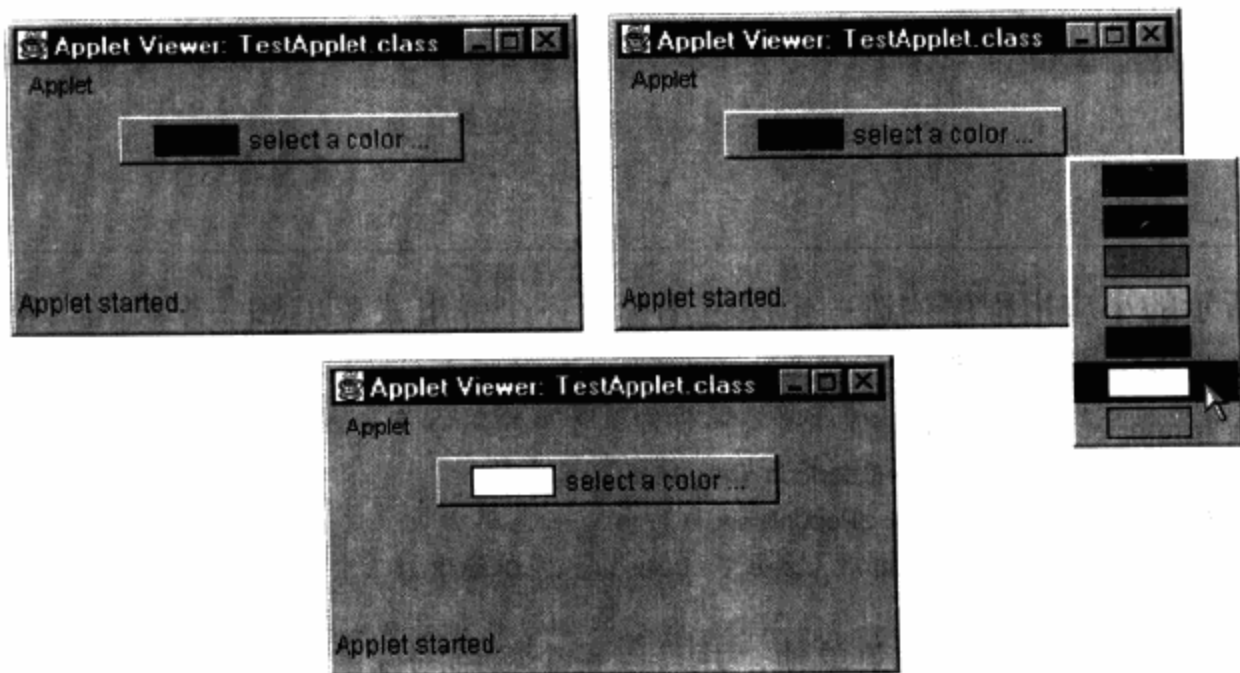


图 5-9 共享图标

图 5-9 所示的小应用程序最让人感兴趣的地方是：它只有一个 `ColorIcon` 实例，按钮和菜单项共享这个实例。

由于 `ColorIcon` 的实例被图 5-9 所示的小应用程序所共享，所以必须对例 5-8 所列的最初的 `ColorIcon` 类进行一个地方的修改，即例 5-8 中是在构造时带入一种填充色，如果代表不同的颜

色的各组件共享图标，则这个填充色需要是一个更动态的属性。

例 5-10 所列的 ColorIcon 版本希望把 Swing 组件传送给它的 paintIcon 方法以便维护一个填充色属性。当绘制图标时，它提取组件的填充色属性并把组件的填充色设置为它的填充色，了解客户属性，请参见 4.9 节“客户属性”。

例 5-10 修改后的 ColorIcon 类清单

```
import java. awt. * ;
import javax. swing. * ;

public class ColorIcon implements Icon {
    private int w, h;

    public ColorIcon (int w, int h) {
        this.w = w;
        this.h = h;
    }

    public void paintIcon (Component c, Graphics g, int x, int y) {
        Color fillColor = Color.lightGray;

        g.setColor (Color.black);
        g.drawRect (x, y, w-1, h-1);

        if (c instanceof JComponent) {
            JComponent jc = (JComponent) c;
            fillColor = (Color) jc.getClientProperty ("fill color");
        }

        g.setColor (fillColor);
        g.fillRect (x+1, y+1, w-2, h-2);
    }

    public int getIconWidth () {
        return w;
    }

    public int getIconHeight () {
        return h;
    }
}
```

图 5-9 所示的小应用程序构造一个颜色图标、一个弹出式菜单和一个按钮。把图标传送给按钮的构造方法会使图标在按钮中显示。

```
public class TestApplet extends JApplet {
    private ColorIcon colorIcon = new ColorIcon (40, 15);
    private JPopupMenu popup = new JPopupMenu ();
    private JButton button = new JButton ("select a color ...", colorIcon);
}
```

接着，实例化弹出式菜单的菜单项，并且以一个颜色图标的引用为参数。为每个菜单项设置适当的填充色属性和一个动作监听器。

```
...
private void addPopupMenuItems () {
    JMenuItem redItem = new JMenuItem (colorIcon),
    blueItem = new JMenuItem (colorIcon),
    grayItem = new JMenuItem (colorIcon),
    yellowItem = new JMenuItem (colorIcon),
}
```

```

        blackItem = new JMenuItem (colorIcon),
        whiteItem = new JMenuItem (colorIcon),
        orangeItem = new JMenuItem (colorIcon);

MenuItemListener listener = new MenuItemListener ();

redItem.putClientProperty ("fill color", Color.red);
redItem.addActionListener (listener);
popup.add (redItem);

blueItem.putClientProperty ("fill color", Color.blue);
blueItem.addActionListener (listener);
popup.add (blueItem);

// add color propety and action listeners for the
// the rest of the menu items ...

```

当一个菜单项激活时，listener 从这个菜单项中提取填充色属性。按钮的填充色被设置为这个菜单项的填充色，然后重新绘制这个按钮。

```

class MenuItemListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JComponent jc = (JComponent) e.getSource ();
        button.putClientProperty ("fill color",
            jc.getClientProperty ("fill color"));
        button.repaint ();
    }
}

```

当该按钮重新绘制后，它调用颜色图标的 paintIcon 方法并传送它自己作为要绘制图标的组件。颜色图标从这个按钮提取填充颜色，并根据这个填充色绘制它自己。

例 5-11 列出了这个小应用程序的完整代码。

例 5-11 在许多组件中共享单个图标的小应用程序

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private ColorIcon colorIcon = new ColorIcon (40, 15);
    private JPopupMenu popup = new JPopupMenu ();
    private JButton button = new JButton ("select a color ...",
                                           colorIcon);

    public void init () {
        addPopupMenuItems ();

        button.putClientProperty ("fill color", Color.red);
        Container cP = getContentPane ();
        CP.setLayout (new FlowLayout ());
        CP.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                Dimension buttonSz = button.getSize ();
                popup.show (button, buttonSz.width,
                    buttonSz.height);
            }
        });
    }
}

```

```

    });
}
private void addPopupMenuItems () {
    JMenuItem redItem = new JMenuItem (colorIcon),
    blueItem = new JMenuItem (colorIcon),
    grayItem = new JMenuItem (colorIcon),
    yellowItem = new JMenuItem (colorIcon),
    blackItem = new JMenuItem (colorIcon),
    whiteItem = new JMenuItem (colorIcon),
    orangeItem = new JMenuItem (colorIcon);

    MenuItemListener listener = new MenuItemListener ();
    redItem.putClientProperty ("fill color", Color.red);
    redItem.addActionListener (listener);
    popup.add (redItem);

    blueItem.putClientProperty ("fill color", Color.blue);
    blueItem.addActionListener (listener);
    popup.add (blueItem);

    grayItem.putClientProperty ("fill color", Color.gray);
    grayItem.addActionListener (listener);
    popup.add (grayItem);

    yellowItem.putClientProperty ("fill color", Color.yellow);
    yellowItem.addActionListener (listener);
    popup.add (yellowItem);

    blackItem.putClientProperty ("fill color", Color.black);
    blackItem.addActionListener (listener);
    popup.add (blackItem);

    whiteItem.putClientProperty ("fill color", Color.white);
    whiteItem.addActionListener (listener);
    popup.add (whiteItem);

    orangeItem.putClientProperty ("fill color", Color.orange);
    orangeItem.addActionListener (listener);
    popup.add (orangeItem);
}
class MenuItemListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JComponent jc = (JComponent) e.getSource ();

        button.putClientProperty ("fill color",
            jc.getClientProperty ("fill color"));

        button.repaint ();
    }
}

```

5.2.3 图像图标

如前所述，Swing 提供 Icon 接口的一个实现，即 ImageIcon 类，类总结 5-2 对这个类进行了总结。

类总结 5-2 ImageIcon

扩展：Object

实现: Icon、Serializable

1. 构造方法

```
public ImageIcon ()
public ImageIcon (byte [])
public ImageIcon (byte [], String description)
public ImageIcon (Image)
public ImageIcon (Image, String description)
public ImageIcon (String filename)
public ImageIcon (String filename, String description)
public ImageIcon (URL)
public ImageIcon (URL, String description)
```

ImageIcon 类提供许多构造方法。可以用一个字节数组、代表图像文件名的字符串、或一个 URL 来指定图像。还提供了用于可访问性的可选的描述。

不带参数的 ImageIcon 构造方法实例化一个没有图像或描述的图像图标。图像或描述可以在构造后再设置。

2. 方法

(1) Icon 接口方法

```
public int getIconHeight ()
public int getIconWidth ()
public synchronized void paintIcon (Component, Graphics, int x, int y)
```

上面列出的方法在 Icon 接口中定义。getIconHeight 和 getIconWidth 方法分别返回图标的高度和宽度。PaintIcon 方法调用 Graphics.drawImage(), 并把传送给这个方法的组件指定为图像观察器。结果, 当把动画 GIF 图像作为一个图像图标显示时, 动画 GIF 图像将动画。

(2) 描述/图像

```
public String getDescription ()
public void setDescription (String)

public Image getImage ()
public void setImage (Image)

public ImageObserver getImageObserver ()
public void setImageObserver (ImageObserver)
```

除了实现在 Icon 接口中定义了的方法外, ImageIcon 还包括一些维护一个读写描述的方法。这个描述是为可访问性所使用的; 例如, 对视力不好的用户, 这个描述可能以音频的形式呈现出来。ImageIcon 还提供设置和获取它显示的图像的方法。

因为 AWT 图像是异步装载, 所以 AWT 装载图像总是有些艰难。通常, 要么用一个 MediaTracker 实例来确保在一幅图像绘制以前这幅图像已完全装载, 要么用一个 ImageObserver 实现来追踪装载的过程。ImageIcon 采用前一种方法——当一幅图像被指定时 (不论在构造时还是在构造后的任何时间), 就调用 MediaTracker 的一个实例以便完全装载这幅图像。图像装载封装在 protected ImageIcon.loadImage 方法中, 因此, 如果需要, ImageIcon 的扩展可任意修改缺省图像装载政策。因为 ImageIcon 同步地装载图像, 所以建议使用小图像。

如果在装载过程中发生了一个错误, 则由图像图标维护的图像设置为 null, 接着调用的 paintIcon() 并不产生任何操作, 而且对 getIconWidth() 和 getIconHeight() 的调用将返回 -1。图像装载的状态可以通过调用 getImageLoadStatus() 来获取, getImageLoadStatus() 返回一个 Me-

diaTracker 常量。

ImageIcon 还提供设置和获取与图像相关联的图像观察器的方法。如果没有显式地设置观察器，那么，当绘制图像时，就把传送给 paintIcon 方法的组件指定为图像观察器。在实践中，开发人员很少关心图像观察器，尤其考虑到通过一个 MediaTracker 实例就能完全装载图像。

图 5-10 示出了显示一个图像图标的小应用程序。

例 5-12 所列的小应用程序简单地实例化一个 ImageIcon 实例，并在重载的 paint 方法中绘制这个图标。

例 5-12 有一个 ImageIcon 的小应用程序

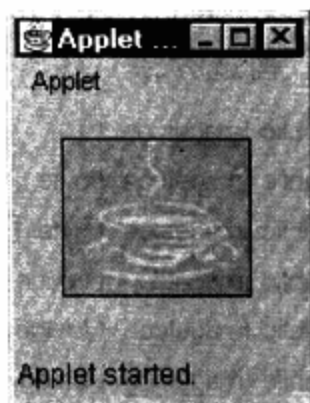


图 5-10 一个图像图标

```
import java.awt.* ;
import javax.swing.* ;

public class Test extends JApplet {
    ImageIcon icon = new ImageIcon ("coffeeCup.jpg");

    public void paint (Graphics g) {
        icon.paintIcon (this, g, 20, 15);
    }
}
```

5.2.4 动画的图像图标

虽然图像图标能被 MediaTracker 的一个实例完全装载，但是，多帧的图像需要一个图像观察器以便显示图像连续的帧。有关图像观察器在多帧图像中的作用的更多信息，请参见《Java 2 图形设计，卷 I：AWT》。

如果通过调用 ImageIcon.setImageObserver () 没有为一个图像图标显式地设置一个图像观察器，则把传送给图标的 paintIcon 方法的组件作为图像观察器。因为 java.awt.Component 是一个能够绘制多帧图像的图像观察器，所以动画的图像会由 ImageIcon 类自动处理。

图 5-11 示出了一个显示动画的 GIF 图像的小应用程序。在有趣的空间里，动画的某些帧没有被显示出来，但是所显示的帧也足以使你对动画的图像图标有一个大概的了解。

例 5-13 列出了图 5-11 所示的小应用程序的代码。

例 5-13 带一个动画的图标的小应用程序

```
import java.awt.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public void init () {
        JPanel panel = new MyJPanel ();
        getContentPane ().add (panel, "Center");
    }
}

class MyJPanel extends JPanel {
    ImageIcon animatedIcon = new ImageIcon ("globe.gif");

    public void paintComponent (Graphics g) {
```

```
super.paintComponent (g);
animatedIcon.paintIcon (this, g, 20, 20);
```

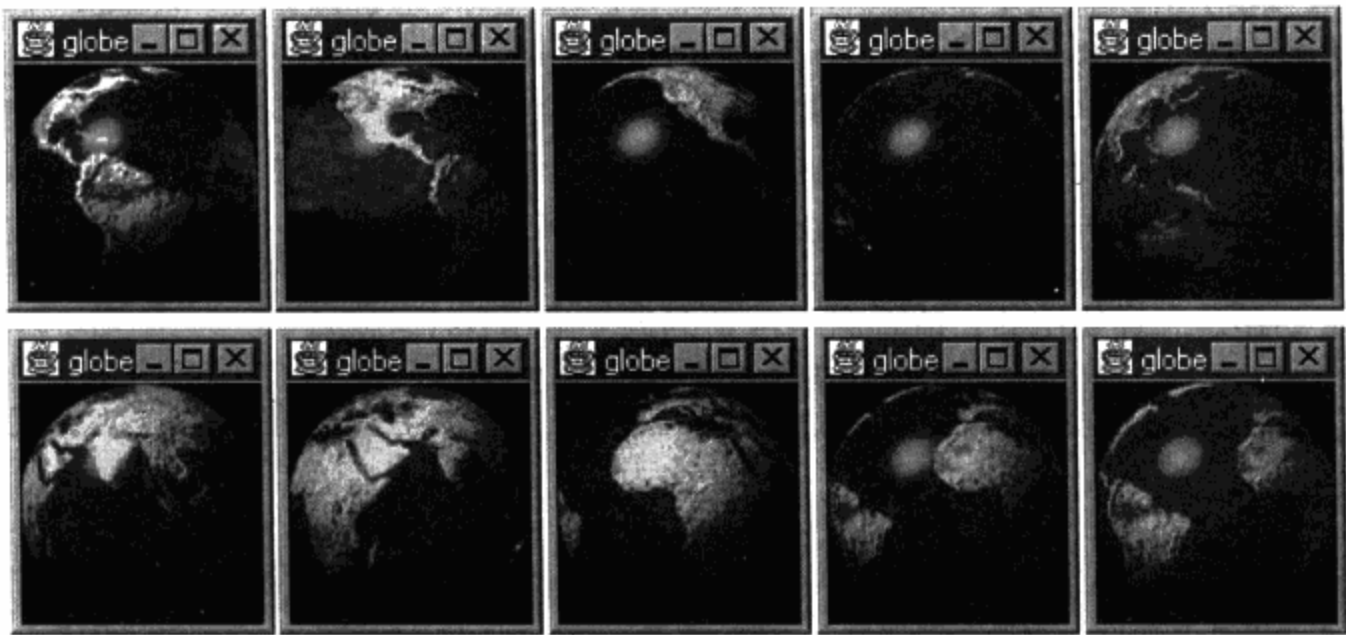


图 5-11 一个动画的图像图标

这个图标没有直接绘制到这个小应用程序中，因为 JApplet 类不是从 JComponent 中派生的，因此，不能双缓存它的显示；如果不使用双缓存，则这个动画将闪烁。相反，JPanel 是从 JComponent 中派生的，缺省时能双缓存它的显示，因此，这个图标显示在一个居中的 JPanel 实例中。

注意，因为 JPanel 是双缓存的，所以传送给图标的 paintIcon 方法的 Graphics 不是这个组件的屏上的 Graphics。取而代之的是，传送给 MYJPanel.paint 方法、接着传送给图标的 paintIcon 方法的 Graphics 是一个屏外缓存。

5.3 动作

动作封装一个与小应用程序或应用程序有关的基础操作。动作除维护一个或多个值（通常是图标和/或字符串）外，还维护一个允许状态。一个动作的基本功能由 swing.Action 接口来定义。接口总结 5-3 列出了 Action 接口（它扩展 java.awt.event.ActionListener 接口）方法。

接口总结 5-3 Action

1. 常量

```
public static final String DEFAULT
public static final String LONG DESCRIPTION
public static final String NAME
public static final String SHORTDESCRIPTION
public static final String SMALL ICON
```

由 Action 接口定义的常量在 5.3.2 节“动作常量”中介绍。

2. 方法

```
public abstract void addPropertyChangeListener (PropertyChangeListener)
public abstract void removePropertyChangeListener (PropertyChangeListener)
public abstract void putValue (String, Object)
```



```
public abstract Object getValue (String)
public abstract boolean isEnabled ()
public abstract void setEnabled (boolean)
```

当动作的属性被修改后，动作将激发属性变化事件。结果，Action 接口为正在登记的属性变化监听器定义方法。动作维护可以被 putValue 和 getValue 方法操纵的字符串/对象对。动作还维护一个允许状态，因此，它提供了 setEnabled 方法和 isEnabled 方法。

Swing 包提供一个实现 Action 接口的 AbstractAction 类，该类实现 Action 接口并提供接口总结 5-3 所列的所有方法的缺省功能。AbstractAction 类是抽象的，因为它不实现由 ActionListener 接口定义的唯一的方法——actionPerformed()。ActionPerformed 方法留给扩展去实现。

要了解动作是怎样使用的，请看例 5-14 所列的小应用程序。

例 5-14 带一个菜单条的小应用程序

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");
        JMenuItem exitItem = new JMenuItem ("exit");

        exitItem.addActionListener (new ExitListener ());
        fileMenu.add (exitItem);
        mb.add (fileMenu);
        setJMenuBar (mb);
    }
}

class ExitListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        System.exit (0);
    }
}
```

例 5-14 的小应用程序创建一个菜单项，并把这个菜单项添加到“file”菜单中。一个 ActionListener 扩展被实现并被添加到这个菜单项中。当“exit”菜单项被激活时，就调用 ExitListener.actionPerformed() 方法，导致了小应用程序的终止。

现在，让我们看看例 5-15 所列的用动作实现的相同的小应用程序。

例 5-15 用一个动作创建一个菜单项

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");
        fileMenu.add (new ExitAction ());
    }
}
```

```

        mb.add (fileMenu);
        setJMenuBar (mb);
    }

    class ExitAction extends AbstractAction {
        public ExitAction () {
            super ("exit");
        }

        public void actionPerformed (ActionEvent e) {
            System.exit (0);
        }
    }

```

例 5-15 的小应用程序与例 5-14 的小应用程序功能完全相同。然而，它没有使用创建一个菜单项并把这个菜单项添加到“file”菜单中的办法，而是实现一个 `AbstractAction` 扩展并把这个扩展添加到“file”菜单中。“file”菜单提取与动作相关联的字符串“exit”，并创建一个将添加到“file”菜单中的菜单项。另外，“file”菜单把动作添加到由菜单项维护的 `ActionListeners` 的列表中。结果，当这个菜单项激活时，则调用 `ExitAction.actionPerformed` 方法，小应用程序终止。

5.3.1 作为控制中心点的动作

就像图标只是绘制在例 5-7 的小应用程序上一样，动作也只是用来创建菜单项。然而，动作真正的价值（与图标一样）是它们具有与多个组件相关联的能力。

如前所述，动作维护一个允许状态，而且可以激发属性变化事件。另外，单个动作可以与多个 `Swing` 组件相关联。当一个动作被启用或禁用时，它发送一个属性变化事件给与之相关联的所有组件。这些组件依次设置它们的允许状态以便与这个动作的状态一致。因此，动作可以被用来作为退出小应用程序、打开一个新文档、剪切和粘贴等逻辑操作的一个控制点。

图 5-12 所示的小应用程序实现一个“保存”操作。这个操作由一个菜单项和一个工具条按钮来代表。这个菜单项和工具条按钮都只用一个 `SaveAction` 实例来构造。当“action enabled”复选框激活时，则相应地设置动作的状态。这引起动作把一个属性变化事件发送给工具条按钮和菜单项。这个属性变化事件引起这个菜单项和工具条按钮更新它们的允许状态以便与动作的允许状态相匹配。图 5-12 左边的图片显示动作被启用，而右边的图片显示动作禁用。

`SaveAction` 类扩展 `AbstractAction`，并把“save”字符串和“保存”图标传送给 `AbstractAction` 构造方法。`SaveAction` 还实现所要求的 `actionPerformed` 方法，这个方法显示触发“保存”操作的组件。

```

class SaveAction extends AbstractAction {
    public SaveAction () {
        super ("save", new ImageIcon ("save.gif"));
        setEnabled (false);
    }

    public void actionPerformed (ActionEvent event) {
        String s = new String ();
        Object o = event.getSource ();

        if (o instanceof JButton)      s += "ToolBar:";
        else if (o instanceof JMenuItem) s += "MenuBar:";

        System.out.println (s + " save");
    }
}

```

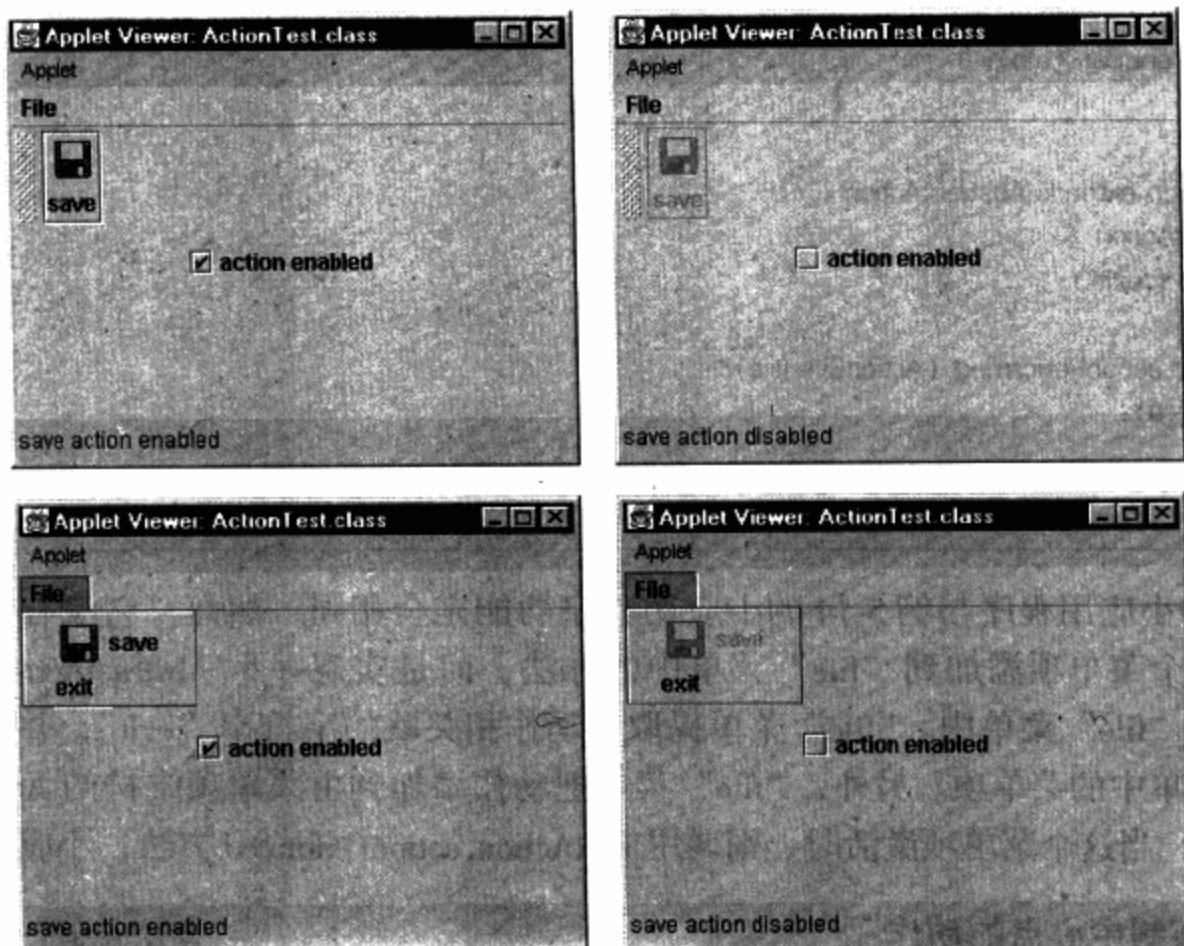


图 5-12 动作：一个控制中心点

这个小应用程序构造菜单条、“file”菜单、工具条和复选框。“保存”动作添加到“file”菜单中，“file”菜单创建一个有字符串和图标（它们从“保存”动作中提取）的菜单项。“保存”动作还添加到工具条中，工具条也提取“保存”动作中的字符串和图标，并创建一个工具条按钮。小应用程序添加自己作为“保存”动作的属性变化监听器，以便当“保存”动作的启用状态改变时，这个小应用程序能得到通知。

```
public class Test extends JApplet
    implements PropertyChangeListener {
    ...
    JCheckBox jc = new JCheckBox ("action enabled");
    JMenuBar mb = new JMenuBar ();
    JToolBar tb = new JToolBar ();

    Action saveAction = new SaveAction ();
    Action exitAction = new ExitAction ();

    public void init () {
        JMenu fileMenu = new JMenu ("File");

        fileMenu.add (saveAction);
        fileMenu.add (exitAction);
        tb.add (saveAction);

        mb.add (fileMenu)
        saveAction.addPropertyChangeListener (this);
        ...
    }
    ...
}
```

当复选框激活时，则触发了“保存”动作的允许属性状态。

```
jc.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
```

```
saveAction.setEnabled (! saveAction.isEnabled ());
```

```
};
```

当 `itemStateChanged` 方法调用 `setEnabled()` 时，一个属性变化事件发送给这个菜单项、这个工具条按钮和这个小应用程序。这个菜单项和工具条按钮更新它们的允许状态，并且，小应用程序显示触发“保存”操作的组件。

例 5-16 列出了图 5-12 所示的小应用程序的代码。

例 5-16 与一个工具条按钮和一个菜单项相关联的动作

```
import java.awt.* ;
import java.awt.event.* ;
import java.beans.* ;
import javax.swing.* ;

public class Test extends JApplet
    implements PropertyChangeListener {
    JPanel jp = new JPanel ();
    JPanel cp = new JPanel (); // cp = checkbox panel
    JCheckBox jc = new JCheckBox ("action enabled");
    JMenuBar mb = new JMenuBar ();
    JToolBar tb = new JToolBar ();

    Action saveAction = new SaveAction ();
    Action exitAction = new ExitAction ();

    public void init () {
        JMenu fileMenu = new JMenu ("File");

        fileMenu.add (saveAction);
        fileMenu.add (exitAction);
        tb.add (saveAction);

        JCheckBoxMenuItem checkBoxItem =
            new JCheckBoxMenuItem ("saved");

        associateActionAndCheckBoxItem (saveAction, checkBoxItem);
        fileMenu.add (checkBoxItem);

        mb.add (fileMenu);

        saveAction.addPropertyChangeListener (this);

        jp.setLayout (new BorderLayout (2, 2));
        jp.add (tb, "North"); // toolbar
        jp.add (cp, "Center"); // checkbox panel

        cp.setLayout (new FlowLayout ());
        cp.add (jc);

        Container contentPane = getContentPane ();

        contentPane.setLayout (new BorderLayout ());
        getRootPane ().setJMenuBar (mb);
        contentPane.add (jp, "Center");

        jc.setSelected (saveAction.isEnabled ());

        jc.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {
                saveAction.setEnabled (! saveAction.isEnabled ());
            }
        });
    }
}
```

```

    }
    });
}

public void propertyChange (PropertyChangeEvent e) {
    boolean b = ( (Boolean) e.getNewValue () ).booleanValue ();
    showStatus ("save action " + (b ? "enabled" : "disabled"));
}

private void associateActionAndCheckBoxItem (
    final Action action,
    final JCheckBoxMenuItem item) {
    item.setHorizontalTextPosition (JButton.LEFT);
    item.setVerticalTextPosition (JButton.CENTER);
    item.setEnabled (action.isEnabled ());
    item.addActionListener (action);
    action.addPropertyChangeListener (
        new PropertyChangeListener () {
            public void propertyChange (PropertyChangeEvent e) {
                String name = e.getPropertyName ();

                if (name.equals (Action.NAME)) {
                    item.setText ( (String) e.getNewValue ());
                    item.revalidate ();
                }

                else if (name.equals ("enabled")) {
                    item.setEnabled (
                        ( (Boolean) e.getNewValue () ).booleanValue ());
                    item.repaint ();
                }

                else if (name.equals (Action.SMALL_ICON)) {
                    item.setIcon ( (Icon) e.getNewValue ());
                    item.revalidate ();
                }
            }
        }
    );
}

class SaveAction extends AbstractAction {
    public SaveAction () {
        super ("save", new ImageIcon ("save.gif"));
        setEnabled (false);
    }

    public void actionPerformed (ActionEvent event) {
        String s = new String ();
        Object o = event.getSource ();

        if (o instanceof JButton) s += "ToolBar:";
        else if (o instanceof JMenuItem) s += "MenuBar:";

        System.out.println (s + " save");
    }
}

class ExitAction extends AbstractAction {
    public ExitAction () {
        super ("exit");
    }

    public void actionPerformed (ActionEvent event) {

```

```
System.exit (0);
```

5.3.2 动作常量

Action 接口还定义动作常量，如表 5-2 所示。

表 5-2 动作常量

方法	含义	对象类型
NAME	在 Swing 组件中显示的一个字符串	字符串
SMALL_ICON	在 Swing 组件中显示的一个图标	图标
DEFAULT	已定义的实现	已定义的实现
LONG_DESCRIPTION	已定义的实现	字符串
SHORT_DESCRIPTION	已定义的实现	字符串

所有常量都是在 Action 类中作为 public static 字符串来定义的。Action.putValue (String, Object) 方法把一个值添加给一个动作。例如，有一个字符串参数和一个图标参数的 AbstractAction 构造方法把 NAME 分配给字符串，把 SMALL_ICON 分配给图标：

```
//Form swing.AbstractAction...
public AbstractAction (String name, Icon icon) {
    this (name);
    putValue (Action.SMALL_ICON, icon);
}
public AbstractAction (String name) {
    putValue (Action.NAME, name);
}
```

当一个 Swing 组件有一个已添加到这个组件中的动作时，常量被用于提取字符串和图标。例如，JMenu.add (Action) 像下面这样提取文本和图标：

```
//Form swing.JMenu...
public JMenuItem add (Action a ) {
    JMenuItem mi = ;
    new JMenuItem ( (String) a.getValue (Action.NAME),
        (Icon) a.getValue (Action.SMALL_ICON));
    ...
}
```

DEFAULT、LONG_DESCRIPTION 和 SHORT_DESCRIPTION 常量不能在 Swing 中使用；它们的含义是已定义的实现。例如，图 5-13 所示的小应用程序包含 JButton 的一个扩展，这个扩展可以用一个动作来构造。SHORT_DESCRIPTION 值用于设置这个按钮的工具提示。

CustomButton 构造方法以一个动作为参数。这个构造方法提取 NAME 和 SMALL_ICON 属性，通过把这些属性传送给 JButton 构造方法来用这些属性为这个按钮设置文本和图标。

还从这个动作中提取 SHORT_DESCRIPTION 属性，

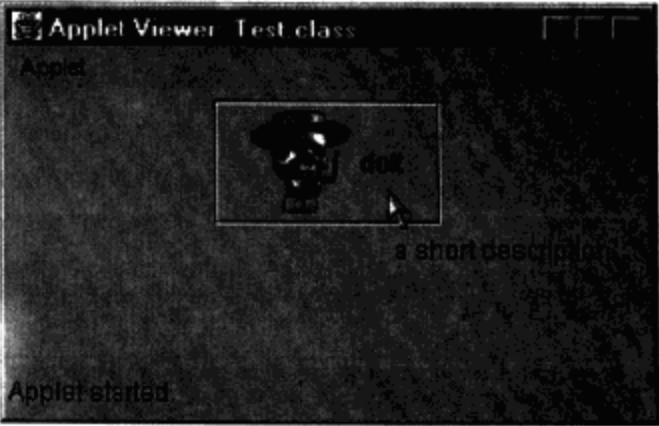


图 5-13 定制组件中所使用的动作

并用 `SHORT_DESCRIPTION` 属性来设置这个按钮的工具提示文本。

最后，构造方法添加这个动作，把它作为这个按钮的一个动作监听器。

```
...
public CustomButton (Action action) {
    super ( (String) action.getValue (Action.NAME),
           (Icon) action.getValue (Action.SMALL_ICON));

    String shortDescription = (String) action.getValue (Action.SHORT_DESCRIPTION);
    setToolTipText (shortDescription);
    addActionListener (action);
}
...
```

例 5-17 列出了图 5-13 所示的小应用程序的代码。

例 5-17 在一个定制组件中使用 `Action.SHORT_DESCRIPTION`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        CustomAction action = new CustomAction ();
        CustomButton button = new CustomButton (action);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);
    }
}

class CustomButton extends JButton {
    public CustomButton (Action action) {
        super ( (String) action.getValue (Action.NAME),
              (Icon) action.getValue (Action.SMALL_ICON));

        String shortDescription = (String) action.getValue (
            Action.SHORT_DESCRIPTION);
        setToolTipText (shortDescription);
        addActionListener (action);
    }
}

class CustomAction extends AbstractAction {
    public CustomAction () {
        super ("doit", new ImageIcon ("skelly.gif"));
        putValue (Action.SHORT_DESCRIPTION, "a short description");
    }

    public void actionPerformed (ActionEvent e) {
        System.out.println ("Custom action performed");
    }
}
```

5.4 本章回顾

边框、图标和动作都有两个共同特点。第一，它们不是组件，第二，支持使用它们的多个

组件可共享它们。为多个组件所共享不仅是一个效率问题，而且还提供了一致性，对动作而言，还提供了 GUI 小应用程序和应用程序的控制中心点。

边框修补了 AWT 容器的一个麻烦，即只能在每个类的基础上指定 AWT 容器的边衬。java.awt.Container 不提供 setInsets 方法，因此，为 AWT 容器设置边衬意味着要子类化类（通常只是为了要重载 getInsets 方法）。为了使同类的不同实例具有不同的属性而强制使用继承不是一种好的面向对象设计，因为这可能导致产生大量的子类。

而 Swing 组件使用空边框来指定它们的边衬。JComponent 类实现一个 setBorder 方法，有效地把只读的 insets 属性提升为读写属性，因此不需要进行子类化。JComponent 重载 getInsets() 以返回其边框（如果有的话）的边衬。如果组件没有边框，它返回其超类的边衬。因此，边框无缝地与 java.awt.Container 类的 insets 属性相结合。不要为你在 Swing 顶层建立的定制组件重载 getInsets()，这样会使所有的好设计失败。相反，让你的定制组件使用一个空边框。

许多界面样式类维护被所有给定类型的组件所共享的单个边框。例如，具有 Windows 界面样式的所有按钮共享单个雕刻边框。如果把边框作为组件来实现，那么共享边框是不可能的。另外，当图标与一个动作相关联时，图标可以为多个组件所共享。

图标和边框另一个极好的特性是它们不需要直接绘制到一个组件中，它们分别绘制到传送给 paintIcon 和 paintBorder 方法的 Graphics 中。这些 paint 方法用传送给它们的组件来获取有关这个组件的信息，但它们在 Graphics 中进行绘制工作。这允许这些 paint 方法的调用者通过指定除直接与这个组件相关联的 Graphics 以外的一个 Graphics 来执行某些处理。例如，如果一个 Swing 组件是双缓存的，则这个组件的边框就绘制到一个屏外缓存中。接着，在合适的时间，屏外缓存拷贝到这个组件的屏上代表中。这样就消除了绘制轻量组件时的闪烁，如果这个边框的 paint 方法只载入一个组件的话，这是不可能的。有关 Swing 的双缓存的更多知识，请参见 4.1.3 节“双缓存”。

动作为逻辑操作提供了一个控制中心点，动作还简化了构造菜单项和工具条按钮等用户界面组件的工作。动作可从一个中心位置来控制各个组件的启用状态，因此，使用动作可以大大简化用户界面代码。

第6章 实用工具

Swing 包括许多实用工具，本章将介绍这些实用工具。其中有些实用工具（如计时器和由 SwingUtilities 类提供的 static 方法）在 Swing 内部使用，而进度监视器和进度监视器流等其他的实用工具则不是内部使用的。使用 Swing 的开发人员可以使用本章介绍的所有实用工具。

6.1 计时器

当某些动作必须周期性地执行或在指定时间内执行时，就要使用计时器。例如，动画可以使用计时器来更新动画的下一帧并控制移动动画对象和改变动画对象外观的速率。要了解使用计时器^①的完整的动画系统（它使用计时器的方式与上面所述的方式完全相同），请参见《Java2 图形设计，卷I：AWT》。事实上，Swing 本身使用计时器来进行自动滚动和显示工具提示。

Swing 提供一个 Timer 类，Timer 类的实例可以有一个或多个与它们相关联的动作监听器。当一个计时器“振铃”时，意味着它激发了一个动作事件，即每个与计时器相关联的 ActionListener 激发了它的 actionPerformed 方法。

可以设置计时器只振铃一次，或按指定的时间间隔重复振铃。每个计时器可以有两种延迟（以毫秒为单位）。Initial delay（初始延迟）指定在计时器第一次振铃之前所需时间量。periodic delay（定期延迟）指定计时器两次振铃之间所需时间量。缺省时，所有的计时器都是重复计时器。

例 6-1 所列的应用程序创建三个计时器。OneSecondTimer 以 1 秒的时间间隔重复振铃，并且与（这个应用程序的）ActionListener 相关联。当 OneSecondTimer 振铃时，它调用这个应用程序的 actionPerformed 方法，该方法显示已计时的秒数。

应用程序创建第二个计时器，它以 2 秒的时间间隔重复振铃，并且有 5 秒的初始延迟。与第二个计时器相关联的动作监听器是一个 TimerWithDelayListener 实例，这个实例显示一个消息：带延迟的计时器正在振铃。

第三个计时器不重复振铃而且有 10 秒的初始延迟。与第三个计时器相关联的 ActionListener 是一个 OneTimeListener 的实例，这个实例显示一个通知：“一次”计时器正在振铃。注意，下面的计时器例子没有显示一个窗口而且必须用 Ctrl + C 才能取消这个小应用程序。

例 6-1 使用 Swing 计时器

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test implements ActionListener {
    private int seconds = 1;

    public Test () {
        Timer oneSecondTimer = new Timer (1000, this);
        Timer timerWithInitialDelay = new Timer (2000,
```

① 在《Java 2 图形设计，卷 I：AWT》中使用的计时器不是 Swing 计时器。

```

        new TimerWithDelayListener ());
    Timer oneTimeTimer = new Timer (10000,
        new OneTimeListener ());
    timerWithInitialDelay.setInitialDelay (5000);
    oneTimeTimer.setRepeats (false);
    oneSecondTimer.start ();
    timerWithInitialDelay.start ();
    oneTimeTimer.start ();
}
public void actionPerformed (ActionEvent e) {
    if (seconds == 0)
        System.out.println ("Time: " + seconds + " second");
    else
        System.out.println ("Time: " + seconds + " seconds");
    seconds ++;
}
public static void main (String args []) {
    new Test ();
    while (true);
}

class TimerWithDelayListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        System.out.println ("Timer with Delay Ringing");
    }
}

class OneTimeListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        System.out.println ("One Time Timer Ringing");
    }
}

```

这三个计时器都用 `Timer` 类提供的唯一构造方法来实例化。这个构造方法载入了两个参数。第一个参数指定与这个计时器有关的延迟，第二个参数指定一个监听器，当计时器振铃时，这个监听器将收到通知。

在构造好三个计时器后，有初始延迟的计时器获得由 `Timer.setInitialDelay` 方法指定的初始延迟。因为缺省时计时器是重复计时器，所以，“一次”计时器调用 `Timer.setRepeats (false)` 以指定这个计时器不重复振铃。接着，用 `Timer.start` 方法来启动这三个计时器。

例 6-1 所列的应用程序的输出如下所示：

```

Time: 1 seconds
Time: 2 seconds
Time: 3 seconds
Time: 4 seconds
Timer with Delay Ringing
Time: 5 seconds
Time: 6 seconds
Timer with Delay Ringing
Time: 7 seconds
Time: 8 seconds
Timer with Delay Ringing
Time: 9 seconds

```

```

One Time Timer Ringing
Time: 10 seconds
Timer with Delay Ringing
Time: 11 seconds
Time: 12 seconds
...

```

计时器类

类总结 6-1 对 Timer 类的方法进行了总结。

类总结 6-1 Timer

1. 构造方法

```
public Timer (int delay, ActionListener)
```

Timer 类只提供一个构造方法。计时器必须有一个延迟和至少一个动作监听器，在构造时必须提供这两个参数。

传送给构造方法的 integer 值指定与计时器有关的延迟。如果计时器重复振铃，则在调用构造方法时指定的延迟既代表初始延迟又代表计时器以后振铃之间的延迟。如果计时器不重复振铃，则在调用构造方法时指定的延迟只代表初始延迟。如果非重复振铃的计时器在构造后又用 setInitialDelay 方法来指定初始延迟，或者，如果用 setDelay 方法为重复计时器指定定期延迟，则在调用构造方法时指定的延迟将被新设置的延迟所取代。

例如，例 6-2 所列的应用程序构造一个有 1 秒延迟的计时器。构造后，这个计时器的初始延迟设置为 10 秒，而且调用 setRepeats (false)，以便计时器不重复振铃。因为在构造后才指定初始延迟而且这个计时器不重复振铃，所以构造计时器时指定的 1 秒的延迟是没有用的，一旦计时器启动 10 秒后，计时器才振铃，而且将不再振铃。

例 6-2 重载构造计时器时所指定的延迟

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test implements ActionListener {
    public Test () {
        Timer oneSecondTimer = new Timer (1000, this);

        oneSecondTimer.setInitialDelay (10000);
        oneSecondTimer.setRepeats (false);
        oneSecondTimer.start ();
    }

    public void actionPerformed (ActionEvent e) {
        System.out.println ("ring ...");
    }

    public static void main (String args []) {
        new Test ();
        while (true);
    }
}

```

2. 方法

(1) 日志计时器

```
public static boolean getLogTimers ()
public static void setLogTimers (boolean)
```

可以记录计时器，即当计时器振铃时，就会把一个消息输出到 System.out 流中。只能通过调用 static setLogTimers 方法来同时为所有的计时器设置日志；没有为一个单独的计时器启动日志的选项。

缺省时，日志消息输出字符串“Timer ringing (计时器振铃):”后跟计时器的 toString 方法的返回值。例如，例 6-3 所列的应用程序调用 Timer.setLogTimers () 来启动日志。

例 6-3 计时器日志

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test implements ActionListener {
    public Test () {
        Timer.setLogTimers (true);

        Timer oneSecondTimer = new MyTimer (1000, this);
        oneSecondTimer.start ();
    }

    public void actionPerformed (ActionEvent e) {
        System.out.println ("ring ...");
    }

    public static void main (String args []) {
        new Test ();
        while (true);
    }
}

class MyTimer extends Timer {
    public MyTimer (int delay, ActionListener listener) {
        super (delay, listener);
    }

    public String toString () {
        return "MyTimer";
    }
}
```

Timer 类不实现 toString 方法，因此，缺省时，Timer 日志输出由 Object.toString 返回的值，这个值表示内存中计时器的位置。如果使用计时器日志，则扩展计时器类及实现一个有意义的 toString 方法都是很重要的，就像例 6-3 所列的应用程序那样。

例 6-3 所列的应用程序的输出如下所示：

```
Timer ringing: MyTimer
ring ...
Timer ringing: MyTimer
ring...
...
```

(2) 启动/停止/重新启动

```
public void start ()
```



```
public void stop ()
public void restart ()
```

可以启动、停止和重新启动计时器。如果一个计时器停止然后又重新启动，则这个计时器在重新启动后将经过整个初始延迟后才会第一次振铃。

(3) 监听器

```
Public void addActionListener (ActionListener)
Public void removeActionListener (ActionListener)
Protected void fireActionPerformed (ActionEvent)
```

可以有多个动作监听器与单个计时器相关联，而且在构造计时器后，可以从这个计时器的监听器列表中删除这些动作监听器。

fireActionPerformed 方法调用每个与这个计时器有关的、登记了的监听器的 actionPerformed。例如，例 6-4 所列的应用程序把三个动作监听器与单个计时器相关联。

例 6-4 与单个计时器相关联的多个动作监听器

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test implements ActionListener {
    private int seconds = 1;

    public Test () {
        Timer oneSecondTimer = new Timer (1000, this);
        oneSecondTimer.addActionListener (new SecondListener ());
        oneSecondTimer.addActionListener (new ThirdListener ());
        oneSecondTimer.start ();
    }

    public void actionPerformed (ActionEvent e) {
        if (seconds == 0)
            System.out.println ("Time: " + seconds + " second");
        else
            System.out.println ("Time: " + seconds + " seconds");
        seconds + + ;
    }

    public static void main (String args []) {
        new Test ();
        while (true);
    }
}

class SecondListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        System.out.println ("Second Listener");
    }
}

class ThirdListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        System.out.println ("Third Listener");
    }
}
```

例 6-4 所列的应用程序的输出如下：

```
Third Listener
Second Listener
Time: 1 second
Third Listener
Second Listener
Time: 2 seconds
Third Listener
Second Listener
Time: 3 seconds
```

注意 `fireActionPerformed` 方法是 `protected`，因此，可以由扩展所重载，这些扩展可以改变通知监听器的顺序。

传送给动作监听器的 `actionPerformed` 方法的 `ActionEvent` 参数的事件源是正在振铃的计时器。

(4) 延迟

```
public int getDelay ()
public int getInitialDelay ()

public void setDelay (int milliseconds)
public void setInitialDelay (int milliseconds)
```

一个计时器的初始延迟和重复延迟在构造后都是可设置的。另外，`Timer` 类提供获取计时器的延迟的方法。

(5) 合并/重复/运行

```
public void setCoalesce (boolean)
public void setRepeats (boolean)

public boolean isCoalesce ()
public boolean isRepeats ()
public boolean isRunning ()
```

`Timer` 类提供用于判断计时器是否重复振铃或是否当前正在运行的方法。还提供了一个设置计时器是否重复振铃的方法。

缺省时，计时器合并事件。如果一个小应用程序或应用程序很忙，不能及时处理由计时器产生的事件，则计时器将把未解决的事件合并为一个通知。例如，例 6-5 所示的应用程序显式地关掉合并功能，因此，应用程序将一个接一个地接收 10 个通知。如果注释掉了 `setCoalesce (false)` 的语句，则应用程序只接收一个事件。

例 6-5 合并计时器事件

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test implements ActionListener {
    private boolean firstRing = true;
    private int ring = 1;

    public Test () {
        Timer.setLogTimers (true);
        Timer oneSecondTimer = new Timer (1000, this);

        // comment out the following line for coalescing
        oneSecondTimer.setCoalesce (false);
    }
}
```

```

        System.out.println ("Timer is coalescing: " +
                             oneSecondTimer.isCoalesce ());

        oneSecondTimer.start ();
    }

    public void actionPerformed (ActionEvent e) {
        System.out.println ("ring #" + ring++);

        if (firstRing) {
            // simulate a time consuming operation by sleeping
            // for 10 seconds ...
            try {
                Thread.currentThread ().sleep (10000);
            }
            catch (InterruptedException ex) {
                ex.printStackTrace ();
            }
            firstRing = false;
        }
    }

    public static void main (String args []) {
        new Test ();
        while (true);
    }
}

```

6.2 事件监听器列表

swing.event 包括一个 `EventListenerList` 类，这个类用来维护一个不同类型的事件监听器列表。在效果上，Swing 的 `EventListenerList` 类替代了 AWT 的 `AWTEventMulticaster` 类，`EventListenerList` 的实例为所有激发事件的 Swing 组件所使用。`EventListenerList` 的实例可以用于任何类型的事件监听器，而 `AWTEventMulticaster` 只限于 AWT 所支持的事件监听器类型集所使用。

鼓励开发人员使用 `EventListenerList` 类来维护激发事件的定制组件（或任何其他类）的事件监听器的列表。使用 `EventListenerList` 类必须遵循下面三个步骤：

- 1) 实现一个把监听器添加到列表中的 `addXXXListener` 方法。
- 2) 实现一个从列表中删除监听器的 `removeXXXListener` 方法。
- 3) 实现一个在列表上循环并把事件发送给监听器的 `fireXXXPerformed` 方法。

Swing 的计时器使用一个 `EventListenerList` 实例来维护一个 `ActionListeners` 列表。下面所列的代码来自 `swing.Timer` 类：

```

// Form swing.Timer...

public void addActionListener (ActionListener listener) {
    listenerList.add (ActionListener.class, listener);
}

public void removeActionListener (ActionListener listener) {
    listenerList.remove (ActionListener.class, listener);
}

protected void fireActionPerformed (ActionEvent e) {
    // Guaranteed to return a non-null array
    Object [] listeners = listenerList.getListenerList ();
    // Process the listeners last to first, notifying

```

```
// those that are interested in this event
for (int i = listeners.length-2; i >= 0; i -= 2) {
    if (listeners[i] == ActionListener.class) {
        ((ActionListener) listeners[i+1]).actionPerformed(e);
    }
}
```

Timer 类的 addActionListener 和 removeActionListener 方法简单地把工作交给 EventListenerList 的 add 和 remove 方法去做。EventListenerList 类的 add 和 remove 方法都有两个参数：第一个参数是监听器的类，第二个参数是对要添加或要删除的监听器的一个引用。

EventListenerList 类维护一个 Object 数组，这个数组包含一个 [Class, ActionListener] 交替序列，它在同一个列表中维护不同类型的监听器。这也是 TimerfireActionPerformed 方法在 for 循环中以 2 为步长遍历数组中的监听器的原因。

注意 Timer.addActionListener 和 Timer.removeActionListener 是不同步的。因为 EventListenerList 类是线程安全的，所以它的 add 和 remove 方法是同步的，因此，Timer 的方法不需要同步。

fireActionPerformed 几乎总是 protected 的。激发动作事件是 Timer 类的实现细节；其他对象应该不允许调用计时器的 fireActionPerformed 方法，因此，这个方法不是 public，而是 protected，以便 Timer 类的扩展可以修改动作事件激发的方式。另外，对激发事件的所有 Swing 组件而言，对监听器列表的遍历从添加到列表中的最后监听器移动到列表中的第一个监听器。

对迫使监听器列表从后向前遍历的 EventListenerList 类没有特别值得介绍之处。事实上，扩展 Timer 类并重载 fireActionPerformed 方法以便从前向后遍历监听器列表是一件简单的事情：

```
// An extension of Timer that reverses the traversal of the
// listener list when action events are fired
class ReverseTimer extends Timer {
    public ReverseTimer (int delay, ActionListener listener) {
        super (delay, listener);
    }
    protected void fireActionPerformed (ActionEvent e) {
        Object [] list = listenerList.getListenerList ();
        // Process the listeners first to last ...
        for (int i = 0; i <= list.length-2; i += 2) {
            if (list[i] == ActionListener.class) {
                ((ActionListener) list[i+1]).actionPerformed(e);
            }
        }
    }
}
```

EventListenerList 类

类总结 6-2 列出了由 EventListenerList 类实现的 public 方法。

类总结 6-2 swing. EventListenerList

扩展：java.lang.Object

实现：java.io.Serializable

1. 构造方法

```
public EventListenerList ()
```

这个无参数的构造方法是由编译器产生的，并且什么也不做。

2. 方法

(1) 添加/删除监听器

```
public synchronized void add (Class, EventListener)
```

```
public synchronized void remove (Class, EventListener)
```

add 和 remove 方法都以监听器类和要添加或要删除的监听器为参数。EventListenerList 类的使用者必须确保传送给这两个方法的类与监听器类是相同的。

(2) 监听器计数

```
public int getListenerCount ()
```

```
public int getListenerCount (Class)
```

getListenerCount () 返回当前列表中监听器的数量。getListenerCount (Class) 返回当前列表中类型是 Class 的监听器的数量。

(3) 监听器列表/字符串代表

```
public Object [] getListenerList ()
```

```
public String toString ()
```

getListenerList () 返回拥有监听器类和监听器本身的实际的 Object 数组。由 getListenerList () 返回的这个数组不会为 null。如果当前的列表中没有监听器，则这个数组的长度等于 0。为了更有效，返回的这个数组不是一个拷贝，因此，不应该由调用者以任何方式修改。

EventListenerList.toString () 显示每个监听器的类型，及列表中当前监听器的数量。

6.3 Swing 实用工具

Swing 包括一个 SwingUtilities 类，这个类有 30 多个 static 实用方法。这些方法按功能分成多个组，如下所示：

- 计算方法。这些方法除了计算特定字体规格的字符串的宽度外，还计算两个矩形间的交集、并集和差集。
- 转换方法。这些方法把一种组件坐标系统中的事件、点和矩形转换为另一种组件坐标系统中的事件、点和矩形。
- 可访问方法。这些方法返回给定组件的可访问信息。
- 检索方法。这些方法检索与给定组件有关的对象，如焦点拥有者、原型和组件所在的根窗格等。
- 从事件派发线程以外的其他线程执行代码的方法。
- 用来确定相互关系的 boolean 方法。例如，一个组件是另一个组件的原型吗？当前的线程是事件派发线程吗？给定一个特定的鼠标事件，哪个鼠标按钮被按下了？
- 一个给定组件的布局、绘制和 UI 代表方法。

由 SwingUtilities 类提供的这些方法在 Swing 中获得了广泛的应用。例如，SwingUtilities.layoutCompoundLabel 由 JButton 和 JLabel 使用以便分别布局与一个按钮或一个标签有关的文本和图标。对使用 Swing 的开发人员来说，这些方法都是很有用的，因此，这些方法都是 public 方法。

类总结 6-3 列出了由 SwingUtilities 类实现的这些 public 方法。

类总结 6-3 SwingUtilities

扩展: java.lang.Object

实现: SwingConstants

1. 构造方法

```
public SwingUtilities ()
```

这个无参数构造方法是由编译器产生的。因为所有由 SwingUtilities 类实现的方法都是 static 方法，所以千万不要实例化 SwingUtilities 的实例。事实上，最好实现一个 private 无参数的构造方法以便 SwingUtilities 的实例不会被实例化。

2. 方法

(1) 计算方法

```
public static Rectangle [] computeDifference (Rectangle, Rectangle)
public static Rectangle computeIntersection (int x, int y, int w, int h, Rectangle)
public static Rectangle computeUnion (int x, int y, int w, int h, Rectangle)
public static final boolean isRectangleContainingRectangle (Rectangle, Rectangle)
public static int computeStringWidth (FontMetrics, String)
```

上面所列的这些方法计算两个矩形之间的并集、交集和差集，还确定一个矩形是否包含了另一个矩形。

图 6-1 所示的小应用程序说明了 computeDifference 方法、computeIntersection 方法和 computeUnion 方法的使用。

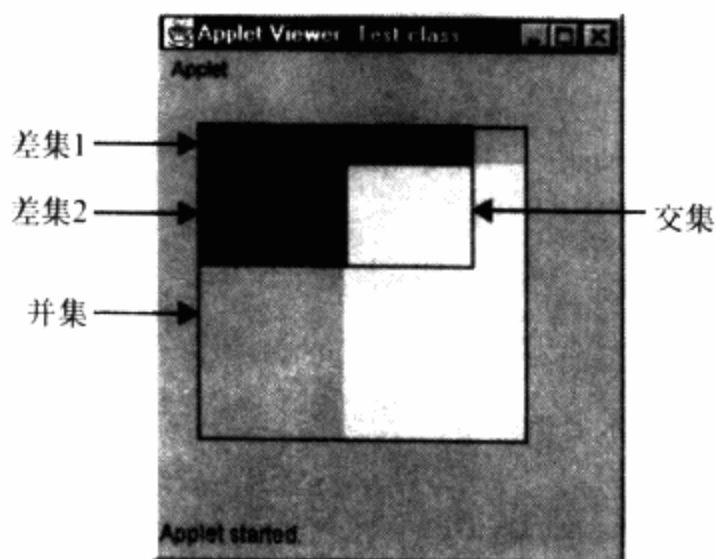


图 6-1 计算两个矩形的差集、交集和并集

computeIntersection 和 computeUnion 都重载传送给它们的矩形值，以便更新 Rectangle 不需要被实例化。当这些方法返回时，传送给这些方法的矩形代表这两个矩形区域的交集或并集。

computeDifference 方法返回一个矩形数组，这个矩形数组代表第一个矩形中没有被第二个矩形所重迭的矩形区域。

例 6-6 列出了图 6-1 所示的小应用程序的代码。

例 6-6 计算两个矩形之间的差集、交集和并集

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
```



```

Rectangle r1 = new Rectangle (20, 20, 150, 75);
Rectangle r2 = new Rectangle (100, 40, 100, 150);
Rectangle destination;

public Test () {
    destination = new Rectangle (r2);

    // print out the intersection of r1 and r2 ...
    // computeUnion stores the union of r1 and
    // destination in destination

    System.out.println ("Intersection: " +
        SwingUtilities.computeIntersection (r1.x, r1.y,
            r1.width, r1.height, destination));
    System.out.println ();

    // print out the union of r1 and r2 ...
    // computeUnion stores the union of r1 and
    // destination in destination

    System.out.println ("Union: " +
        SwingUtilities.computeUnion (r1.x, r1.y,
            r1.width, r1.height, destination));
    System.out.println ();

    // print out the difference of r1 and r2 ...
    // computeDifference does not stuff return values
    // in an input argument

    Rectangle [] difference =
        SwingUtilities.computeDifference (r1, r2);

    System.out.println ("Difference:");
    for (int i=0; i < difference.length; ++i) {
        System.out.println (difference [i]);
    }
}

public void paint (Graphics g) {
    g.setColor (Color.red);
    g.fillRect (r1.x, r1.y, r1.width, r1.height);

    g.setColor (Color.yellow);
    g.fillRect (r2.x, r2.y, r2.width, r2.height);
}
}

```

图 6-1 所示的小应用程序的输出如下所示：

Intersection: java.awt.Rectangle [x = 100, y = 40, width = 70, height = 55]

Union: java.awt.Rectangle [x = 20, y = 20, width = 150, height = 75]

Difference:

Java.awt.Rectangle [x = 20, y = 20, width = 150, height = 20]

Java.awt.Rectangle [x = 20, y = 40, width = 80, height = 55]

(2) 转换方法

```

public static MouseEvent convertMouseEvent (Component, MouseEvent, Component)
public static Point convertPoint (Component, int x, int y, Component)
public static Point convertPoint (Component, Point, Component)
public static Rectangle converRectangle (Component, Rectangle, Component)

```

```
public static void convertPointFromScreen (Point, Component)
public static void convertPointToScreen (Point, Component)
```

上面所列的方法把点、矩形、或鼠标事件从一个组件的坐标系转换到另一个组件的坐标系。上面所列的第一组方法都以对两个组件的引用为参数，这两个组件一个是源组件，一个是目的组件。源组件表示位置或发生的事件源自这个组件。目的组件是位置或事件要转换到的目的组件。

最后的两个方法提供方便地从一个组件的坐标系转换到屏幕的坐标系的方法，和从屏幕的坐标系转换到一个组件的坐标系的方法。

图 6-2 所示的小应用程序有三个嵌套的面板，所有这些面板都用一种特定的颜色来绘制它们的背景，并显示相对于它们的坐标系统的鼠标坐标。

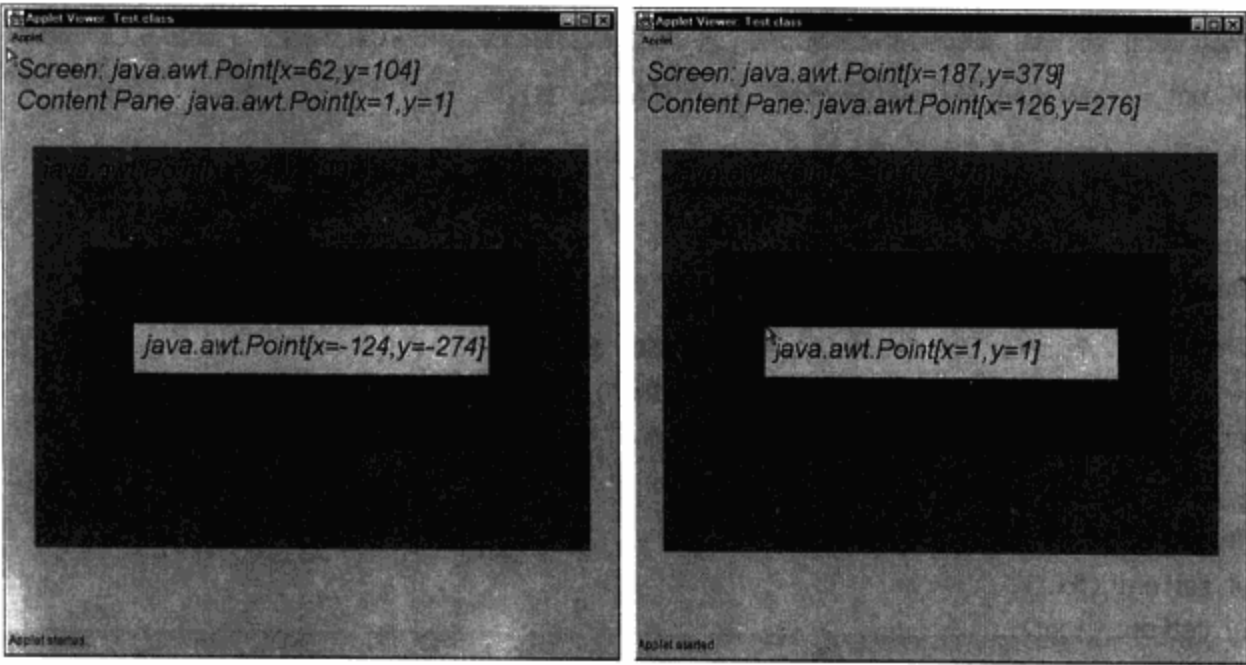


图 6-2 转换坐标系

一个添加到这个小应用程序内容窗格中的鼠标移动监听器把相对于内容窗格坐标系统的鼠标移动事件的位置转换为相对于其他面板坐标系和屏幕坐标系统的鼠标移动事件的位置。

```
...
contentPane.addMouseMotionListener (
    new MouseMotionAdapter () {
public void mouseMoved (MouseEvent e) {
    Point pt = e.getPoint ();

    outer.setString (SwingUtilities.convertPoint (
        contentPane, pt, outer) .toString ());

    inner.setString (SwingUtilities.convertPoint (
        contentPane, pt, inner) .toString ());

    innermost.setString (SwingUtilities.convertPoint (
        contentPane, pt, innermost) .toString ());

    SwingUtilities.convertPointToScreen (
        pt, contentPane);

    lastScreenPt = pt;
    repaint ();
    }
});
...
```

例 6-7 列出了图 6-2 所示的小应用程序的代码。

例 6-7 转换鼠标坐标

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;
public class Test extends JApplet {
    private Point lastScreenPt = null;
    private final Container contentPane = getContentPane ();
    private PanelWithString
        outer = new PanelWithString (Color.orange),
        inner = new PanelWithString (Color.red),
        innermost = new PanelWithString (Color.yellow);

    public Test () {
        Font font = new Font ("Times-Roman", Font.ITALIC, 26);
        contentPane.setLayout (new OverlayLayout (contentPane));
        contentPane.add (innermost);
        contentPane.add (inner);
        contentPane.add (outer);

        innermost.setMaximumSize (new Dimension (350, 50));
        inner.setMaximumSize (new Dimension (450, 200));
        outer.setMaximumSize (new Dimension (550, 400));

        setFont (font);
        innermost.setFont (font);
        inner.setFont (font);
        outer.setFont (font);

        contentPane.addMouseMotionListener (
            new MouseMotionAdapter () {
                public void mouseMoved (MouseEvent e) {
                    Point pt = e.getPoint ();

                    outer.setString (SwingUtilities.convertPoint (
                        contentPane, pt, outer) .toString ());

                    inner.setString (SwingUtilities.convertPoint (
                        contentPane, pt, inner) .toString ());

                    innermost.setString (SwingUtilities.convertPoint (
                        contentPane, pt, innermost) .toString ());

                    SwingUtilities.convertPointToScreen (
                        pt, contentPane);

                    lastScreenPt = pt;
                    repaint ();
                }
            }
        );
    }

    public void paint (Graphics g) {
        super.paint (g);

        if (lastScreenPt != null) {
            String s = new String ("Screen: " + lastScreenPt);

            g.setColor (getForeground ());
            g.drawString (s, 10, g.getFontMetrics () .getHeight ());
        }
    }
}
```

```

SwingUtilities.convertPointFromScreen (lastScreenPt,
                                      contentPane);

s = "Content Pane: " + lastScreenPt;
g.drawString (s, 10, g.getFontMetrics ().getHeight () * 2);
}
else {
    g.setColor (getForeground ());
    g.drawString ("MOVE THE MOUSE IN HERE", 10,
                g.getFontMetrics ().getHeight ());
}
}
}

class PanelWithString extends JPanel {
    String s;
    Color color;

    public PanelWithString (Color color) {
        this.color = color;
    }

    public void setString (String s) {
        this.s = s;
    }

    public void paintComponent (Graphics g) {
        super.paintComponent (g);

        Dimension size = getSize ();

        g.setColor (color);
        g.fillRect (0, 0, size.width, size.height);

        if (s != null) {
            g.setColor (getForeground ());
            g.drawString (s, 10, g.getFontMetrics ().getHeight ());
        }
    }
}

```

(3) 可访问方法

```

public static Accessible getAccessibleAt (Component, Point)
public static Accessible getAccessibleChild (Component, int)
public static int getAccessibleChildrenCount (Component)
public static int getAccessibleIndexInParent (Component)
public static AccessibleStateSet getAccessibleStateSet (Component)

```

上面所列的这些方法用来获取一个给定组件的可访问性信息。

(4) 焦点拥有者/原型/根窗格/窗口

```

public static Component findFocusOwner (Component)
public static Container getAncestorNamed (String, Component)
public static Container getAncestorOfClass (Class, Component)
public static Component getDeepestComponentAt (Component, int, int)
public static Rectangle getLocalBounds (Component)
public static JRootPane getRoot (Component)
public static JRootPane getRootPane (Component)
public static Window windowForComponent (Component)

```

上面所列的这些方法返回与一个特定组件有关的信息。给定一个组件，则可获得与这个组件有关的焦点拥有者、命名了的原型、根窗格、窗口等信息。

(5) 激活可运行对象

```
public static void invokeAndWait (Runnable)
```

弹出 InterruptedException 和 InvocationTargetException 异常信息;

```
public static void invokeLater (Runnable)
```

invokeAndWait 和 invokeLater 方法用于从事件派发线程以外的一个线程中执行代码。这两种方法在“Swing 和线程”中介绍过，因此，这里就不再作介绍了。

(6) 子组件/事件派发线程

```
public static boolean isDescendingFrom (Component allegedDescendent,
```

```
Component allegedAncestor)
```

```
public static boolean isEventDispatchThread ()
```

上面所列的两种方法可以用于确定一个组件是否是另一个组件的子组件，当前的线程是否是事件派发线程。

(7) 鼠标按钮

```
public static boolean isLeftMouseButton (MouseEvent)
```

```
public static boolean isMiddleMouseButton (MouseEvent)
```

```
public static boolean isRightMouseButton (MouseEvent)
```

在 AWT 中，给定一个鼠标事件，要确定是按下了哪个鼠标按钮，就要涉及检查事件的修饰符并把它们与一个位掩码进行“与”，这个过程不直观而且很难记住。Swing 修补了 AWT 的这个缺陷，它提供了三个方便的方法——在已知鼠标事件时能确定哪个鼠标按钮按下了。

(8) 组合标签/绘制/更新组件树 UI

```
public static String layoutCompoundLabel (FontMetrics, String, Icon, int, int, int, int,
```

```
Rectangle, Rectangle, Rectangle, int)
```

```
public String layoutCompoundLabel (JComponent, FontMetrics, String, Icon, int, int, int, int,
```

```
Rectangle, Rectangle, Rectangle, int)
```

```
public static void paintComponent (Graphics, Component, Container, int, int, int, int)
```

```
public static void paintComponent (Graphics, Component, Container, Rectangle)
```

```
public static void updateComponentTreeUI (Component)
```

Swing 按钮和标签用 layoutCompoundLabel 方法布局它们的文本和图标。PaintComponent 方法用于在一个任意图形上绘制一个组件，而 updateComponentTreeUI 方法用于更新与一个组件的子组件相关联的 UI 代表。

6.4 Swing 常量

Swing 在 swing 包中提供了三个定义常量的接口。SwingConstants 类定义位置常量，ScrollPaneConstants 类和 WindowConstants 类为它们各自的组件定义常量。

接口总结 6-1 列出了由 SwingConstants 接口定义的常量。

接口总结 6-1 SwingConstants

常量

```
public static final int BOTTOM
```

```
public static final int CENTER
```

```
public static final int EAST
```

```

public static final int HORIZONTAL
public static final int LEFT
public static final int NORTH
public static final int NORTH EAST
public static final int NORTH WEST
public static final int RIGHT
public static final int SOUTH
public static final int SOUTH EAST
public static final int SOUTH WEST
public static final int TOP
public static final int VERTICAL
public static final int WEST

```

许多 Swing 类实现 `SwingConstants` 接口，这些类是：`JCheckBoxMenuItem`、`JLabel`、`JProgressBar`、`JSlider` 和 `TextField`。实现 `SwingConstants` 接口允许访问由 `SwingConstants` 接口定义的常量。

6.5 BorderLayout 和 Box 类

Swing 提供了一个布局管理器 (`BoxLayout`) 来水平或垂直布局各组件，还提供了一个容器 (`Box`)，这个容器利用一个 `BoxLayout` 类。

6.5.1 BoxLayout 类

`BoxLayout` 沿水平线或垂直线布局组件。在构造时指定一个轴 (`BoxLayout.X_AXIS` 或 `BoxLayout.Y_AXIS`)，并且确定组件的方向。

沿着 x 轴布局的组件为水平布局，沿 y 轴布局的组件为垂直布局。一般情况下按组件添加到容器中的顺序来从上到下或从左到右地布局组件。

`BoxLayout` 分别把水平或垂直布局的组件的大小安排为所想要的宽度或高度。对水平布局而言，如果所有的组件没有相同的高度，则 `BoxLayout` 试图把每个组件的高度设置为容器中最高的组件的高度。如果一个组件的高度不能设置为最高组件的高度，则根据它的垂直排列方式，垂直放置这个组件。在垂直布局时，则以水平布局时调整组件高度同样的方式来调整组件宽度。本节提供一个简单的使用 `BoxLayout` 布局管理器的例子。参见 10.10 节“`JToolBar`”中的一个例子，这个例子更复杂，且考虑了组件的排列方式。

图 6-3 示出了一个小应用程序，它有两个容器，其中每个容器都有一些按钮。这两个容器都用一个 `BoxLay-`

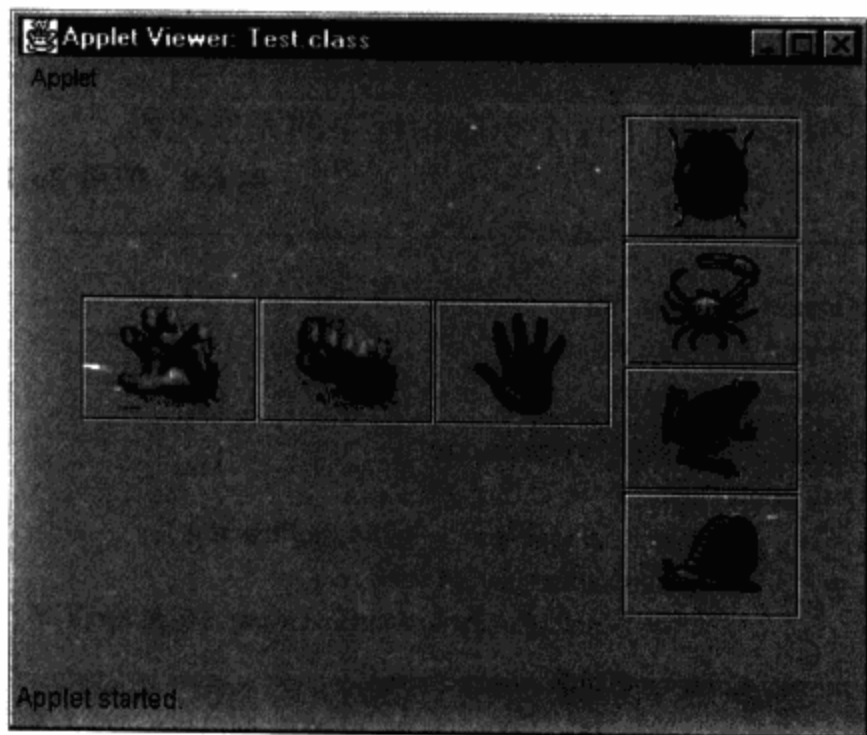


图 6-3 两个使用 `BoxLayout` 的容器

out 作为它们的布局管理器；左边的容器沿着 `X_AXIS` 方向布局组件，右边的容器沿着 `Y_AXIS` 方向布局组件。

图 6-3 所示的小应用程序实现一个 `JPanel` 类扩展，这个扩展使用一个 `BoxLayout` 实例作为它

的布局管理器：

```
class ContainerWithBoxLayout extends JPanel {
    public ContainerWithBoxLayout (int orientation) {
        setLayout (new BoxLayout (this, orientation));
    }
}
```

这个小应用程序实例化两个 `ContainerWithBoxLayout` 实例：一个实例以 `BoxLayout.X_AXIS` 为参数，另一个实例以 `BoxLayout.Y_AXIS` 为参数：

```
public class Test extends JApplet {
    public Test () {
        ...
        ContainerWithBoxLayout yaxis =
            new ContainerWithBoxLayout (BoxLayout.Y_AXIS);
        ContainerWithBoxLayout xaxis =
            new ContainerWithBoxLayout (BoxLayout.X_AXIS);
```

接着，把一些按钮添加到这两个容器中，然后，再把这两个容器添加到小应用程序的内容窗格中。

```
...
xaxis.add (new JButton (new ImageIcon ("reach.gif")));
xaxis.add (new JButton (new ImageIcon ("punch.gif")));
xaxis.add (new JButton (new ImageIcon ("open_hand.gif")));
yaxis.add (new JButton (new ImageIcon ("ladybug.gif")));
yaxis.add (new JButton (new ImageIcon ("crab.gif")));
yaxis.add (new JButton (new ImageIcon ("frog.gif")));
yaxis.add (new JButton (new ImageIcon ("snail.gif")));
contentPane.setLayout (new FlowLayout ());
contentPane.add (xaxis);
contentPane.add (yaxis);
}
```

例 6-8 列出了这个小应用程序的完整代码。

例 6-8 使用 `BoxLayout`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        ContainerWithBoxLayout yaxis =
            new ContainerWithBoxLayout (BoxLayout.Y_AXIS);
        ContainerWithBoxLayout xaxis =
            new ContainerWithBoxLayout (BoxLayout.X_AXIS);
        contentPane.setLayout (new FlowLayout ());
        xaxis.add (new JButton (new ImageIcon ("reach.gif")));
        xaxis.add (new JButton (new ImageIcon ("punch.gif")));
        xaxis.add (new JButton (new ImageIcon ("open_hand.gif")));
        yaxis.add (new JButton (new ImageIcon ("ladybug.gif")));
```

```

yaxis.add (new JButton (new ImageIcon ("crab.gif")));
yaxis.add (new JButton (new ImageIcon ("frog.gif")));
yaxis.add (new JButton (new ImageIcon ("snail.gif")));

contentPane.add (xaxis);
contentPane.add (yaxis);
|
|
class ContainerWithBoxLayout extends JPanel {
    public ContainerWithBoxLayout (int orientation) {
        setLayout (new BoxLayout (this, orientation));
    }
}

```

BoxLayout 是一个简单的布局管理器。与大多数 AWT 布局管理器不同，它不预先设置组件之间的间隙或组件与容器边缘之间的间隙。

6.5.2 Box 类

Box 类与 BoxLayout 类关系最为密切，Box 类是一个 java.awt.Container 的扩展，它用一个 BoxLayout 实例来布局组件。Box 类并没有使 BoxLayout 变得更容易使用，事实上，就减少处理 BoxLayout 的复杂性而言，Box 确实不比例 6-8 所列的 ContainerWithBoxLayout 类做得多。使 Box 类真正闪光的地方是它提供的一些 static 方法，这些方法返回三种组件中的一种，这三种组件是：胶体（glue）、膨胀体（struts）和固定区（rigid areas）。

胶体组件可以认为是胶粘的物质，它们可扩展填充由邻近组件所定义的区域。实际上，使用胶体这个词是不太妥当的，因为“胶体”不固定，而且使用它不会使旁边的组件固定不动。也许更好的类比应该是“橡皮泥”，这是在大多数商店都可买得到的。

膨胀体在所选择的一维空间上是固定的，它们填充另一维空间。例如，BoxLayout.createVerticalStrut 方法载入一个高度参数，这个组件（又称作膨胀体）维护创建这个组件时所使用的高度并尽量扩展这个组件的宽度。

固定区以一个 Dimension 为参数来构造，而且，固定区大小永远不变。例如，如果用 (100,100) 来创建一个固定区，则固定区的大小将总是 (100,100)。

在很多情况下，例如，在设计输入表单时，胶体、膨胀体和固定区都是很有用的。图 6-4 示出了上、下两个带标签和组合框的面板图。上图由 GridBagLayout 布局，而下图使用了水平和垂直的膨胀体，使布局更美观了。

例 6-9 列出了图 6-4 所示的面板的一部分代码。这个面板使用 GridBagLayout 的一个实例作为它的布局管理器，并结合使用了水平和垂直的膨胀体。

例 6-9 使用水平和垂直的膨胀体

```

class AnchorFillWeightPanel extends JPanel {
    ...

```

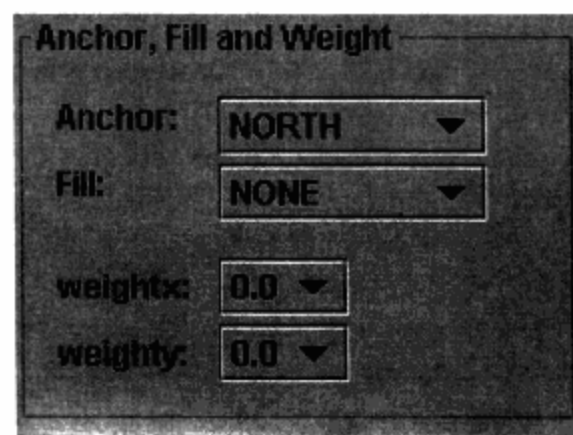
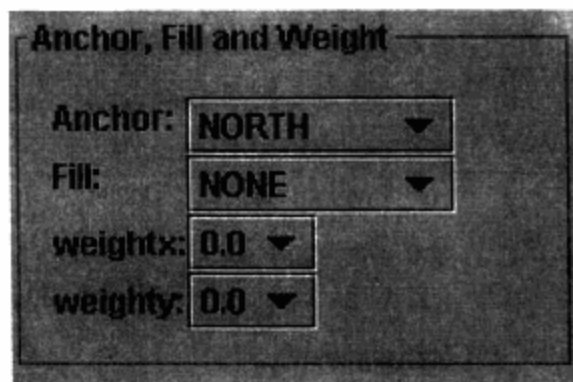


图 6-4 使用膨胀体：上图和下图

```

public AnchorFillWeightPanel () {
    ...
    GridBagLayout gbl = new GridBagLayout ();
    GridBagConstraints gbc = new GridBagConstraints ();
    setLayout (gbl);
    gbc.anchor = GridBagConstraints.NORTHWEST;
    add (anchorLabel, gbc);
    add (Box.createHorizontalStrut (10), gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.weightx = 1.0;
    add (anchorCombo, gbc);
    gbc.weightx = 0;
    add (Box.createVerticalStrut (3), gbc);
    gbc.gridwidth = 1;
    add (fillLabel, gbc);
    add (Box.createHorizontalStrut (10), gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.weightx = 1.0;
    add (fillCombo, gbc);
    gbc.weightx = 0;
    add (Box.createVerticalStrut (13), gbc);
    gbc.gridwidth = 1;
    gbc.anchor = GridBagConstraints.WEST;
    add (weightxLabel, gbc);
    add (Box.createHorizontalStrut (10), gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.weightx = 1.0;
    add (weightxCombo, gbc);
    gbc.weightx = 0;
    add (Box.createVerticalStrut (3), gbc);
    gbc.gridwidth = 1;
    add (weightyLabel, gbc);
    add (Box.createHorizontalStrut (10), gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.weightx = 1.0;
    add (weightyCombo, gbc);
    ...
}

```

虽然 GridBagLayout 看起来有些令人畏惧, 但是, 把它用于与图 6-4 类似的表单实际上是很容易的。交替地用 GridBagConstraints.REMAINDER 和 1 作为网格宽度把组件放入了它们相应的列中。用 GridBagLayout 布局表单的更复杂例子, 请参见 22.1 “JTextField”。

类总结 6-4 总结了 Box 类。

类总结 6-4 Box

扩展: java.awt.Container

实现: javax.accessibility.Accessible

1. 构造方法

```
public Box (int axis)
```

所有的框都用一个 integer 值来构造, 这个 integer 值代表一个轴, 组件将沿着这个轴被布局。有效的 integer 值是:

- BorderLayout.X_AXIS
- BorderLayout.Y_AXIS

2. 方法

(1) 框和框物质

```
public static Box createHorizontalBox ()
public static Box createVerticalBox ()

public static Component createGlue ()
public static Component createHorizontalGlue ()
public static Component createVerticalGlue ()

public static Component createHorizontalStrut (int width)
public static Component createVerticalStrut (int height)
public static Component createRigidArea (Dimension)
```

用 createHorizontalBox 和 createVerticalBox 方法可以方便地创建框, 可见下面的两行代码:

```
Box hb = new Box (Box.X_AXIS);
Box vb = new Box (Box.Y_AXIS);
```

通过控制胶体、膨胀体和固定区的最小、最大和首选尺寸来实现胶体、膨胀体和固定区。

表 6-1 列出了可以用 Box static 方法创建的每个物质类型所需的大小。

表 6-1 框实体的 Min/首选/Max

实体	最小尺寸	首选尺寸	最大尺寸
胶体	0, 0	0, 0	short.MAX_VALUE, short.MAX_VALUE
水平胶体	0, 0	0, 0	short.MAX_VALUE, 0
垂直胶体	0, 0	0, 0	0, short.MAX_VALUE
水平膨胀体	宽度, 0	宽度, 0	宽度, short.MAX_VALUE
垂直膨胀体	0, 高度	0, 高度	short.MAX_VALUE, 高度
固定区	dimension1 ^①	dimension	dimension

① 固定区以三个 dimension 为参数来构造。

应该强调的是: 由 static Box 方法返回的物质其行为可能不象胶体、膨胀体和固定区 (如果它们被添加到一个没有把 BorderLayout 作为其布局管理器的容器中)。其他的布局管理器可能忽略对最小尺寸、首选尺寸和最大尺寸的大小请求。

(2) 布局管理器/可访问相关内容

```
public void setLayout (LayoutManager)
public AccessibleContext getAccessibleContext ()
```

每个框都使用 BorderLayout 的一个实例作为它的布局管理器, 这需要重载 setLayout 方法。

与所有的 Swing 组件一样, 为了可访问性, Box 类实现 getAccessibleContext () 方法, 该方法返回 AccessibleContext 的一个实例。

6.6 进度监视器

对花很长时间来完成的操作应该提供一些可视的指示，指示已完成了百分之多少的任务。通常，这种可视的指示器是一个进度条，它通常显示在允许取消任务的对话框中。

Swing 有一个进度条组件——`JProgressBar`，它用来指示已完成了百分之多少的任务。在第八章“进度条、滑杆和分隔条”中介绍了一个用进度条监视一个费时任务的小应用程序。

为监视进度，Swing 还提供了两个类，它们创建进度条并在对话框中显示进度条，这两个类是：`ProgressMonitor` 和 `ProgressMonitorInputStream`。前者提供设置进度的方法，并更新由监视器创建的进度条。后者是 `java.io.FilterInputStream` 的一个扩展，它除显示一个包含进度条的对话框外（如果正在读的流是费时的操作的话），其使用方式与任何其他输入流的使用方式相同。

6.6.1 ProgressMonitor

`ProgressMonitor` 类通过显示一个进度对话框来监视费时任务的进度。进度监视器在两个属性的帮助下决定一个操作任务所花费的时间是否很长，需要显示进度对话框，这两个属性是：`millisToDecideToPopup` 和 `millisToPopup`，缺省时，它们分别是 500 毫秒和 2000 毫秒（0.5 秒和 2 秒）。这两个属性都是可设置的。

从创建进度监视器时开始，当 `millisToDecideToPopup` 指定的毫秒过去时，进度监视器根据到目前为止已完成任务的百分比来计算完成这个操作所需花费的时间。如果所需的整个时间比 `millisToDecideToPopup` 指定的时间长，则显示一个进度对话框。

使用进度监视器的步骤如下：

- 1) 实例化一个 `ProgressMonitor` 实例。
- 2) 定期调用 `ProgressMonitor.setProgress(int)`（也可选择调用 `ProgressMonitor.setNote(String)`）。
- 3) 当操作完成时，调用 `ProgressMonitor.close()`。

图 6-5 所示的应用程序包含一个按钮，激活这个按钮将开始读取一个文件。这个应用程序使用一个进度监视器来显示读文件的进度。

左上图显示按钮激活时的情况，此时开始读文件。右上图显示当读文件时显示的进度监视器的对话框。左图显示对话框（和任务）正在取消，通过显示一个消息对话框来处理对话框取消操作。

这个应用程序把一个动作监听器添加到这个按钮中，这个动作监听器创建一个缓存输入流和一个进度监视器，接着，这个监听器创建一个 `ReadThread` 实例，这个实例读文件并更新进度监视器。

进度监视器用一个父组件来创建，这个父组件作为监视器的进度对话框的父组件。`ProgressMonitor` 构造方法还载入一个将在进度对话框中显示的消息和一个注释。这个注释在显示对话框时可以更新；例如，可以用“Reading Files”消息来创建进度监视器，而这个监视器的注释可以更新为正在读取的文件名。还可以用代表任务的最小值和最大值来创建进度监视器。

图 6-5 所示的应用程序创建一个 `ProgressMonitor`，它指定应用程序的内容窗格作为父组件、“Reading Files”作为消息、文件名作为注释，指定最小值为 0、最大值为文件的字节数。

```
public class Test extends JFrame {
```

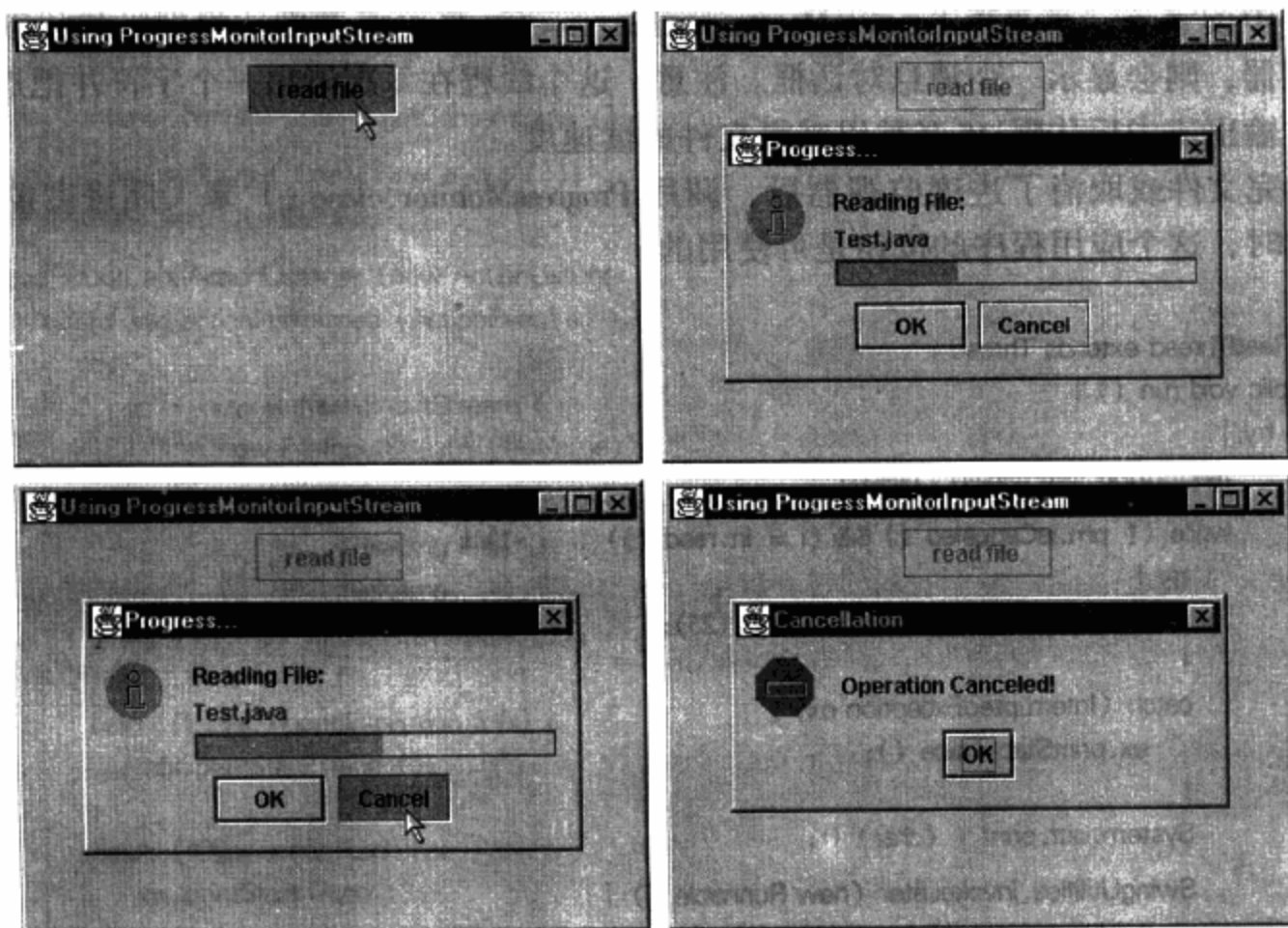


图 6-5 使用一个进度监视器

```

private JButton readButton = new JButton ("read file");
private BufferedInputStream in;
private ProgressMonitor pm;
private String fileName = "Test.java";

public Test () {
    ...
    readButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            try {
                in = new BufferedInputStream (
                    new FileInputStream (fileName));

                pm = new ProgressMonitor (contentPane,
                    "Reading File:",
                    fileName,
                    0, in.available ());

            }
            catch (FileNotFoundException fnfx) {
                fnfx.printStackTrace ();
            }
            catch (IOException iox) {
                iox.printStackTrace ();
            }

            ReadThread t = new ReadThread ();
            t.start ();

        }
    });
    ...
}

```

ReadThread 类的 run 方法把按钮的启用状态设置为 false。只要进度监视器还没有取消，就

能从文件中读字节，并调用 `ProgressMonitor.setProgress()` 来更新进度监视器。如果取消了这个进度监视器，则会显示一个消息对话框。注意，这个线程在每次读出一个字符并把这个字符输出到标准输出流中后休眠 25 毫秒以减慢文件的读速度。

在读完文件或取消了进度监视器后，调用 `ProgressMonitor.close()` 来关闭进度监视器的对话框，此时，这个应用程序的按钮是可使用的。

```
...
class ReadThread extends Thread {
    public void run () {
        try {
            readButton.setEnabled (false);
            while (! pm.isCanceled () && (i = in.read ()) != -1) {
                try {
                    Thread.currentThread ().sleep (25);
                }
                catch (InterruptedException ex) {
                    ex.printStackTrace ();
                }
                System.out.print ( (char) i);
                SwingUtilities.invokeLater (new Runnable () {
                    public void run () {
                        pm.setProgress ( ++ cnt);
                    }
                });
            }
            if (pm.isCanceled ()) {
                JOptionPane.showMessageDialog (
                    Test.this,
                    "Operation Canceled!",
                    "Cancellation",
                    JOptionPane.ERROR_MESSAGE);

                pm.close ();
            }
            catch (IOException ex) {
                ex.printStackTrace ();
            }
            readButton.setEnabled (true);
        }
    }
}
```

例 6-10 列出了图 6-5 所示的应用程序的完整代码。

例 6-10 使用一个进度监视器

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

public class Test extends JFrame {
    private JButton readButton = new JButton ("read file");
    private BufferedInputStream in;
    private ProgressMonitor pm;
```

```

private String fileName = "Test.java";

public Test () {
    final Container contentPane = getContentPane ();
    contentPane.setLayout (new FlowLayout ());
    contentPane.add (readButton);

    readButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            try {
                in = new BufferedInputStream (
                    new FileInputStream (fileName));

                pm = new ProgressMonitor (contentPane,
                    "Reading File:",
                    fileName,
                    0, in.available ());

            }
            catch (FileNotFoundException fnfx) {
                fnfx.printStackTrace ();
            }
            catch (IOException iox) {
                iox.printStackTrace ();
            }

            ReadThread t = new ReadThread ();
            t.start ();
        }
    });
}

class ReadThread extends Thread {
    int i, cnt = 0;
    String s;

    public void run () {
        try {
            readButton.setEnabled (false);

            while (! pm.isCanceled () && (i = in.read ()) != -1) {
                try {
                    Thread.currentThread ().sleep (25);
                }
                catch (InterruptedException ex) {
                    ex.printStackTrace ();
                }

                System.out.print ( (char) i);

                SwingUtilities.invokeLater (new Runnable () {
                    public void run () {
                        pm.setProgress (+ + cnt);
                    }
                });
            }

            if (pm.isCanceled ())
                JOptionPane.showMessageDialog (
                    Test.this,
                    "Operation Canceled!",
                    "Cancellation",

```

```

        JOptionPane.ERROR_MESSAGE);
        pm.close ();
    }
    catch (IOException ex) {
        ex.printStackTrace ();
    }
    readButton.setEnabled (true);
}

public static void main (String args []) {
    GJApp.launch (new Test (),
        "Using Progress Monitors", 300, 300, 450, 300);
}

```

类总结 6-5 总结了 ProgressMonitor 类。

类总结 6-5 ProgressMonitor

扩展: java.lang.Object

1. 构造方法

public ProgressMonitor (Component parentComponent, Object message, String note, int minimum, int maximum)

如前所述, ProgressMonitor 的实例以一个父组件、一个消息、一个注释、一个最小值和最大值为参数来创建。

2. 方法

public void close ()

public int getMaximum ()

public int getMillisToDecideToPopup ()

public int getMillisToPopup ()

public int getMinimum ()

public String getNote ()

public boolean isCanceled ()

public void setMaximum (int)

public void setMillisToDecideToPopup (int)

public void setMillisToPopup (int)

public void setMinimum (int)

public void setNote (String)

public void setProgress (int)

ProgressMonitor 提供一个 close 方法, 它关闭监视器的进度对话框。如果进度监视器用一个比监视器的最大值还大的值调用它的 setProgress 方法, 则监视器本身就调用这个 close 方法。

ProgressMonitor 类还提供一些获取和设置 minimum、maximum、note、millisToPopup 和 millisToDecideToPopup 属性的方法。

6.6.2 ProgressMonitorInputStream

ProgressMonitorInputStream (java.io.FilterInputStream 的一个扩展) 创建一个进度监视器以监视流的读取。ProgressMonitorInputStream 的实例的使用方式与其他输入流的使用方式相同。

本节介绍一个应用程序，它除使用 `ProgressMonitorInputStream` 来读文件外，其他功能与图 6-5 所示的应用程序完全相同。由于它们的相似性，所以就不显示本节讨论的应用程序了。

与图 6-5 所示的应用程序一样，这个应用程序有一个开始读文件的按钮。应用程序把一个动作监听器添加到这个按钮中，这个监听器除创建读文件的一个 `ReadThread` 实例外，还创建一个 `ProgressMonitorInputStream` 实例。

```
public class Test extends JFrame {
    private ProgressMonitorInputStream in;
    private JButton readButton = new JButton ("read file");
    public Test () {
        ...
        readButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                try {
                    in = new ProgressMonitorInputStream (
                        contentPane,
                        "Reading " + fileName,
                        new FileInputStream (fileName));
                }
                catch (FileNotFoundException ex) {
                    ex.printStackTrace ();
                }

                ReadThread t = new ReadThread ();
                readButton.setEnabled (false);
                t.start ();
            }
        });
    }
    ...
}
```

`ReadThread` 方法简单地读文件，它在读之间有 10 毫秒的休眠以放慢读文件的速度。如果由进度监视器输入流显示的进度对话框被取消，则这个输入流将抛出一个异常，这个异常由应用程序通过显示一个消息对话框来处理。

```
...
class ReadThread extends Thread {
    public void run () {
        int i;
        try {
            while ( (i = in.read ()) != -1) {
                System.out.print ( (char) i);
                try {
                    Thread.currentThread ().sleep (10);
                }
                catch (Exception ex) {
                    ex.printStackTrace ();
                }
            }
            in.close ();
        }
        catch (IOException ex) {
            JOptionPane.showMessageDialog (
                Test.this,
```



```

        catch (Exception ex) {
            ex.printStackTrace ();
        }
    }
    in.close ();
}
catch (IOException ex) {
    JOptionPane.showMessageDialog (
        Test.this,
        "Operation Canceled!",
        "Cancellation",
        JOptionPane.ERROR_MESSAGE);
}
readButton.setEnabled (true);
}
}
public static void main (String args []) {
    GJApp.launch (new Test (),
        "Using ProgressMonitorInputStream",
        300, 300, 450, 300);
}
}

```

类总结 6-6 总结了 ProgressMonitorInputStream。

类总结 6-6 ProgressMonitorInputStream

扩展: java.io.FilterInputStream

1. 构造方法

```
public ProgressMonitorInputStream (ComponentParent Component, Object message,
                                   InputStream)
```

ProgressMonitorInputStream 的实例以一个父组件、一个消息和一个输入流为参数来创建。与 ProgressMonitor 实例的消息一样，传送给 ProgressMonitorInputStream 构造方法的消息是一个 Object 引用。处理 ProgressMonitorInputStream 消息的方式与处理选项窗格消息的方式相同，有关选项窗格和消息的详细内容，请参见 14.3 节“JOptionPane”。

2. 方法

```
public ProgressMonitor getProgressMonitor ()
```

```
public void close () 弹出 IOException 异常信息
```

```
public int read () 弹出 IOException 异常信息
```

```
public int read (byte []) 弹出 IOException 异常信息
```

```
public int read (byte [], int, int) 弹出 IOException 异常信息
```

```
public synchronized void reset () 弹出 IOException 异常信息
```

```
public long skip (long) 弹出 IOException 异常信息
```

ProgressMonitorInputStream 提供了引用流的进度监视器方法。其余由 ProgressMonitorInputStream 类实现的方法都从 java.io.FilterInputStream 类中重载以维护这个流的进度监视器。

6.7 撤消/重复

Swing 提供支持撤消和重复操作的类和接口，这些类和接口在 javax.swing.undo 包中定义，

它代表一般的撤消/重复工具。javax.swing.undo 包的一个类图如图 6-6 所示。

可撤消的操作（又称编辑）由 UndoableEdit 接口来表示。javax.swing.undo 包提供四个实现 UndoableEdit 接口的类，它们是 AbstractUndoableEdit（别管它的名字，它不是抽象类）、CompoundEdit、UndoManager 和 StateEdit。还提供 UndoableEditSupport 类以援助可撤消编辑的通知监听器。

在接口总结 6-2 中总结了 UndoableEdit 接口。

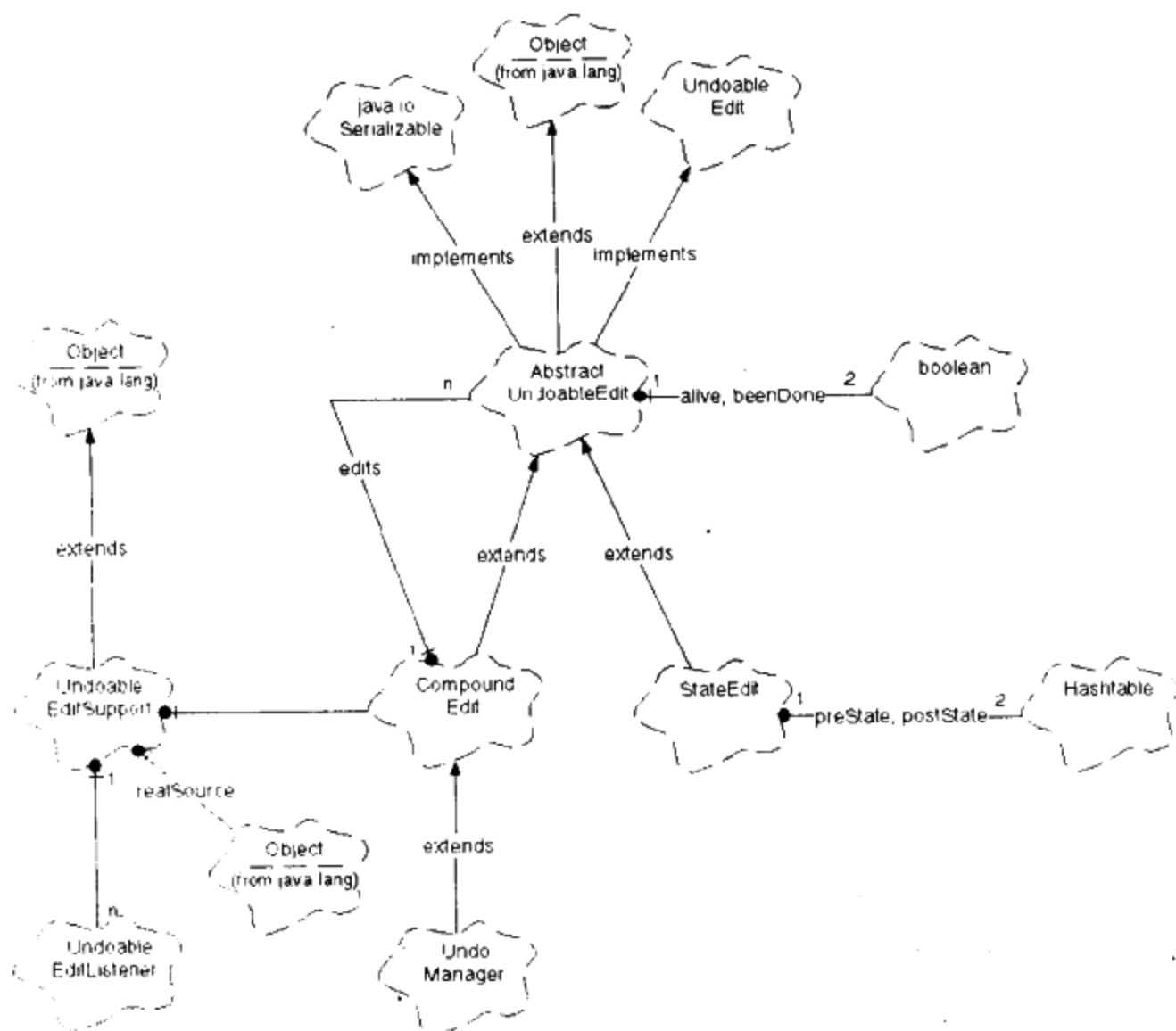


图 6-6 javax.swing.undo 包

接口总结 6-2 UndoableEdit

1. 撤消/重复

```
public abstract boolean canRedo ()
public abstract boolean canUndo ()
public abstract void redo () 弹出 CannotRedoException 异常信息
public abstract void undo () 弹出 CannotUndoException 异常信息
public abstract void die ()
```

通过实现由 UndoableEdit 接口定义的 canRedo 和 canUndo 方法，可撤消编辑必须能够报告它们是否可撤消或重复一个操作。undo 和 redo 方法分别执行撤消和重复操作。注意，如果在一个操作不能撤消或重复时分别调用了 undo 和 redo 方法，则这两个方法都可能弹出异常信息。要使编辑动作不可以再撤消或重复，就要调用 die 方法。可撤消编辑动作通常使用 die 方法来释放与一个操作相关联的资源。

2. 展示名

```
public abstract String getPresentationName ()
public abstract String getRedoPresentationName ()
public abstract String getUndoPresentationName ()
```

每个编辑动作都必须能够报告一个展示名、一个撤消展示名和一个重复展示名。一个编辑动作的展示名应该提供一个代表这个编辑动作的、本地化的、人们可读的名字。撤消展示名代表对这个编辑动作的可撤消形式的一种描述，重复展示名代表对这个编辑动作的可重复形式的一种描述。这三个名字通常都以某种形式呈现给用户。例如，在一个菜单项中显示的文本。

3. 合并编辑

```
public abstract boolean addEdit (UndoableEdit)
public abstract boolean replaceEdit (UndoableEdit)
```

可以通过使一次编辑动作吸收另一次编辑动作来合并可撤消编辑动作。AddEdit 方法载入的编辑动作将引起调用这个方法的编辑动作吸收。replaceEdit 方法与 addEdit 方法正好相反，引起调用 replaceEdit 方法的编辑动作将传送给这个方法的编辑动作吸收。

4. 重要性

```
public abstract boolean isSignificant ()
```

可以把编辑动作指定为重要的或不重要的。不重要的编辑动作通常是一个重要编辑动作的副效果；例如，当选择文本接着又删除文本时，文本选择很可能是一个不重要的编辑动作。

通常用编辑动作的重要性来确定哪些编辑动作要呈现给用户，而且重要性还被 UndoManager 所使用，参见 6.7.4 节“UndoManager”。

6.7.1 一个简单的撤消/重复样例

图 6-7 所示的小应用程序是撤消和重复一个操作的简单的样例。这个小应用程序提供一个菜单，这个菜单包含一个菜单项，这个菜单项允许修改小应用程序所包含的一个面板的背景颜色。这个菜单还提供另一个菜单项，允许撤消和重复背景颜色的改变。

这个小应用程序实现 AbstractUndoableEdit 类的一个扩展——BackgroundColorEdit，而且用

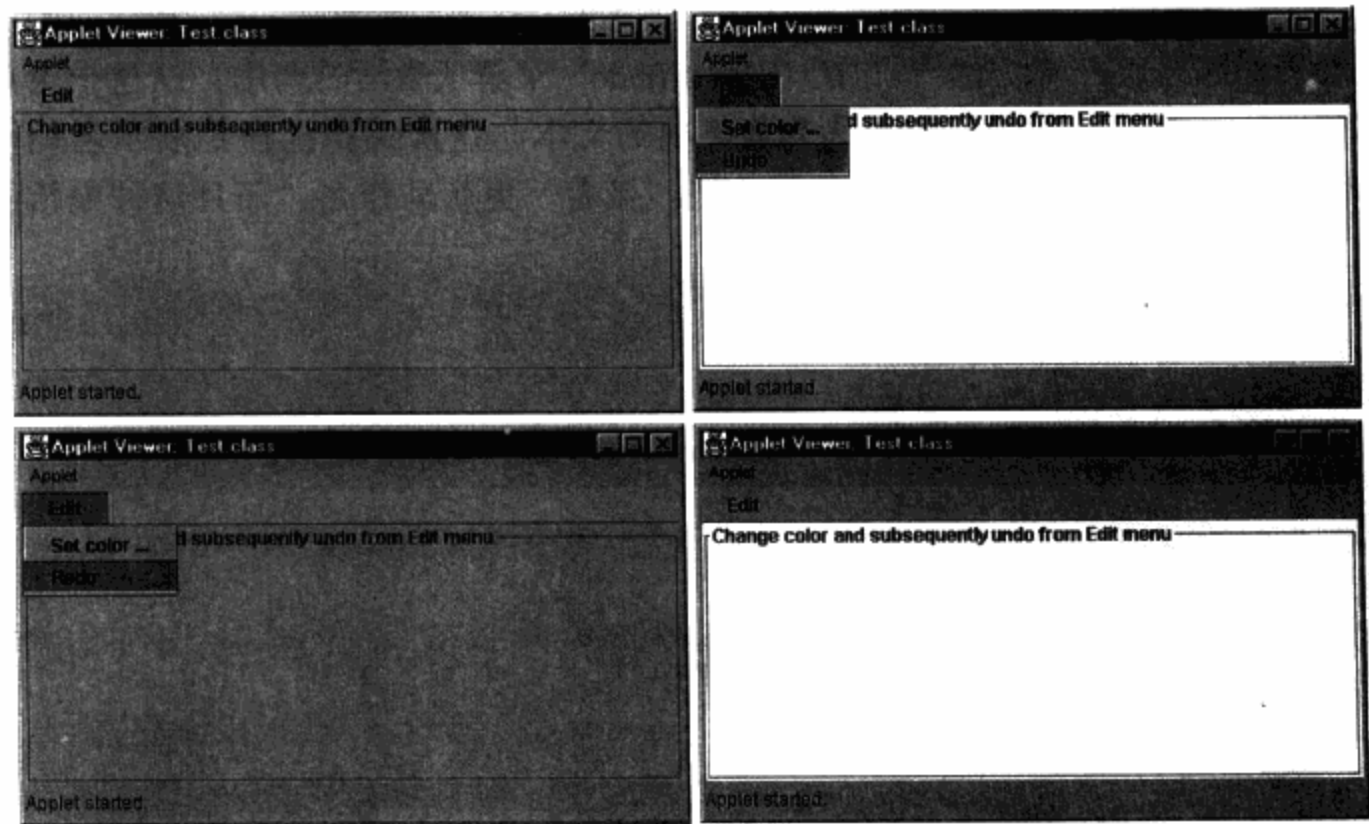


图 6-7 简单的撤消/重复样例

BackgroundColorEdit 的一个实例来撤消和重复背景颜色的变化。这个小应用程序还跟踪以前的（旧的）背景颜色。

```
public class Test extends JApplet {
    private JPanel colorPanel = new JPanel ();
    private BackgroundColorEdit edit = new BackgroundColorEdit ();
    private Color oldColor;
    ...
}
```

这个小应用程序还实现 AbstractAction 类的两个内部类扩展；setColorAction 和 UndoAction，它们分别用来设置背景颜色和撤消/重复这个操作。有关动作和 AbstractAction 类的更多讨论参见 5.3 节“动作”。SetColorAction 的一个实例与“Set color...”菜单项相关联，而 UndoAction 的一个实例与“Undo/Redo”菜单项相关联。

SetColorAction.actionPerformed 方法（它在“Set color...”菜单项激活时调用）显示一个用于选择新的背景颜色的调色板。如果从调色板中选取了一种颜色，则保存当前的背景颜色，并把面板的背景颜色设置为选取的颜色：

```
...
class SetColorAction extends AbstractAction {
    public SetColorAction () {
        super ("Set color ...");
    }
    public void actionPerformed (ActionEvent e) {
        Color color = JColorChooser.showDialog (
            Test.this, // parent component
            "Pick A Color", // dialog title
            null); // initial color
        if (color != null) {
            oldColor = colorPanel.getBackground ();
            colorPanel.setBackground (color);
        }
    }
}
...
```

UndoAction.actionPerformed 方法（它在“Undo/Redo”菜单项激活时调用）从这个菜单项获得文本（用 Action.getValue 方法）。如果这个菜单项的文本与这次编辑的撤消展示名相同，则为这次编辑调用 undo 方法，否则，调用 redo 方法。接着，更新动作名，而且这种更新使菜单项上显示的文本改变了。

```
...
class UndoAction extends AbstractAction {
    public UndoAction () {
        putValue (Action.NAME, edit.getUndoPresentationName ());
    }
    public void actionPerformed (ActionEvent e) {
        String name = (String) getValue (Action.NAME);
        boolean isUndo = name.equals (
            edit.getUndoPresentationName ());
        if (isUndo) {
            edit.undo ();
            putValue (Action.NAME,
                edit.getRedoPresentationName ());
        }
    }
}
```

```

    }
    else {
        edit.redo ();
        putValue (Action.NAME,
            edit.getUndoPresentationName ());
    }
}
...

```

BackgroundColorEdit 类扩展 AbstractUndoableEdit。undo 和 redo 方法都简单地恢复以前的背景色。注意，这两种方法都调用它们超类的同名方法以确保动作的状态保持同步。

```

...
class BackgroundColorEdit extends AbstractUndoableEdit {
    public void undo () throws CannotUndoException {
        super.undo ();
        toggleColor ();
    }
    public void redo () throws CannotRedoException {
        super.redo ();
        toggleColor ();
    }
    public String getUndoPresentationName () {
        return "Undo";
    }
    public String getRedoPresentationName () {
        return "Redo";
    }
    private void toggleColor () {
        Color color = colorPanel.getBackground ();
        ColorPanel.setBackground (oldColor);
        oldColor = color;
    }
}
}

```

例 6-12 列出了图 6-7 所示的小应用程序的完整代码。

例 6-12 一个简单的撤消/重复样例

```

import javax.swing.*;
import javax.swing.undo.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private JPanel colorPanel = new JPanel ();
    private BackgroundColorEdit undo = new BackgroundColorEdit ();
    private Color oldColor;

    public void init () {
        colorPanel.setBorder (
            BorderFactory.createTitledBorder (
                "Change color and subsequently undo " +
                "from the Edit menu"));
        makeMenuBar ();
    }
}

```

```

    getContentPane ().add (colorPanel, BorderLayout.CENTER);
}

private void makeMenuBar () {
    JMenuBar menuBar = new JMenuBar ();
    JMenu editMenu = new JMenu ("Edit");

    editMenu.add (new SetColorAction ());
    editMenu.add (new UndoAction ());

    menuBar.add (editMenu);
    setJMenuBar (menuBar);
}

class SetColorAction extends AbstractAction {
    public SetColorAction () {
        super ("Set color ...");
    }

    public void actionPerformed (ActionEvent e) {
        Color color = JColorChooser.showDialog (
            Test.this, // parent component
            "Pick A Color", // dialog title
            null); // initial color

        if (color != null) {
            oldColor = colorPanel.getBackground ();
            colorPanel.setBackground (color);
        }
    }
}

class UndoAction extends AbstractAction {
    public UndoAction () {
        putValue (Action.NAME, undo.getUndoPresentationName ());
    }

    public void actionPerformed (ActionEvent e) {
        String name = (String) getValue (Action.NAME);
        boolean isUndo = name.equals (
            undo.getUndoPresentationName ());

        if (isUndo) {
            undo.undo ();
            putValue (Action.NAME,
                undo.getRedoPresentationName ());
        }
        else {
            undo.redo ();
            putValue (Action.NAME,
                undo.getUndoPresentationName ());
        }
    }
}

class BackgroundColorEdit extends AbstractUndoableEdit {
    public void undo () throws CannotUndoException {
        super.undo ();
        toggleColor ();
    }

    public void redo () throws CannotRedoException {
        super.redo ();
        toggleColor ();
    }
}

```

```

    |
    |
    | public String getUndoPresentationName () {
    |     return "Undo";
    | }
    |
    | public String getRedoPresentationName () {
    |     return "Redo";
    | }
    |
    | private void toggleColor () {
    |     Color color = colorPanel.getBackground ();
    |     colorPanel.setBackground (oldColor);
    |     oldColor = color;
    | }
    |
    |
    |
    |

```

6.7.2 UndoableEditSupport

图 6-7 所示的样例是一个简单的实现撤消/重复的例子；然而，它不是非常现实的。通常，对撤消/重复的支持内置在组件中。当在一个组件上执行一个可撤消的编辑动作时，这个组件把一个可撤消编辑动作发送给已登记的可撤消编辑动作监听器。例如，例 6-13 重写图 6-7 所示及例 6-12 所列的小应用程序，以便由 ColorPanel 类来创建可撤消编辑并把它发送给监听器。

例 6-13 使用 UndoableEditSupport

```

import javax.swing. * ;
import javax.swing.event. * ;
import javax.swing.undo. * ;
import java.awt. * ;
import java.awt.event. * ;

public class Test extends JApplet {
    private ColorPanel colorPanel = new ColorPanel ();
    private UndoAction undoAction = new UndoAction ();

    public void init () {
        colorPanel.setBorder (
            BorderFactory.createTitledBorder (
                "Change color and subsequently undo " +
                "from the Edit menu"));

        makeMenuBar ();
        colorPanel.addUndoableEditListener (undoAction);
        getContentPane ().add (colorPanel, BorderLayout.CENTER);
    }

    private void makeMenuBar () {
        JMenuBar menuBar = new JMenuBar ();
        JMenu editMenu = new JMenu ("Edit");

        editMenu.add (new SetColorAction ());
        editMenu.add (undoAction);

        menuBar.add (editMenu);
        setJMenuBar (menuBar);
    }

    class UndoAction extends AbstractAction

```



```

        implements UndoableEditListener {
UndoableEdit lastEdit;

public UndoAction () {
    putValue ( Action.NAME, "Undo");
    setEnabled ( false);
}

public void actionPerformed ( ActionEvent e) {
    String name = (String) getValue ( Action.NAME);
    boolean isUndo = name.equals (
        lastEdit.getUndoPresentationName ());
    if (isUndo) {
        lastEdit.undo ();
        putValue ( Action.NAME,
            lastEdit.getRedoPresentationName ());
    }
    else {
        lastEdit.redo ();
        putValue ( Action.NAME,
            lastEdit.getUndoPresentationName ());
    }
}

public void undoableEditHappened ( UndoableEditEvent e) {
    lastEdit = e.getEdit ();
    putValue ( Action.NAME,
        lastEdit.getUndoPresentationName ());

    if (lastEdit.canUndo ())
        setEnabled ( true);
}

}

class SetColorAction extends AbstractAction {
    public SetColorAction () {
        super ("Set color ...");
    }

    public void actionPerformed ( ActionEvent e) {
        Color color = JColorChooser.showDialog (
            Test.this, // parent component
            "Pick A Color", // dialog title
            null); // initial color

        if (color != null) {
            colorPanel.setBackground (color);
        }
    }
}

class ColorPanel extends JPanel {
    UndoableEditSupport support;
    BackgroundColorEdit edit = new BackgroundColorEdit ();
    Color oldColor;

    public void addUndoableEditListener (
        UndoableEditListener l) {
        support.addUndoableEditListener (l);
    }
}

```

```

public void removeUndoableEditListener (
    UndoableEditListener l) {
    support.removeUndoableEditListener (l);
}

public void setBackground (Color color) {
    oldColor = getBackground ();
    super.setBackground (color);

    if (support == null)
        support = new UndoableEditSupport ();

    support.postEdit (edit);
}

class BackgroundColorEdit extends AbstractUndoableEdit {
    public void undo () throws CannotUndoException {
        super.undo ();
        toggleColor ();
    }

    public void redo () throws CannotRedoException {
        super.redo ();
        toggleColor ();
    }

    public String getUndoPresentationName () {
        return "Undo Background Color Change";
    }

    public String getRedoPresentationName () {
        return "Redo Background Color Change";
    }

    private void toggleColor () {
        Color color = getBackground ();
        setBackground (oldColor);
        oldColor = color;
    }
}

```

例 6-13 所列的小应用程序把 BackgroundColorEdit 类作为 ColorPanel 类的一个内部类来实现，而且 ColorPanel 类在它的背景颜色修改时将激发可撤消编辑事件。

UndoAction 类通过实现 undoableEditHappened 方法来实现 UndoableEditListener 接口，它保留对这个编辑动作的一个引用，并根据这个编辑动作是否可以撤消来设置动作的允许状态。

ColorPanel 类在 UndoableEditSupport 类的帮助下激发可撤消编辑事件，它提供用于添加和删除可撤消编辑监听器并把事件发送给监听器的方法。

6.7.3 组合编辑

经常有这种情况：多个可撤消编辑动作都必须立即存储和撤消。CompoundEdit 类（它扩展 AbstractUndoableEdit）提供了这个能力。

下面介绍组合编辑是怎样工作的：把一个组合编辑实例化，并把可撤消编辑动作添加到这个组合编辑中。当正在把编辑动作添加到一个组合编辑中时，这个组合编辑处于进行状态而且直到调用 CompoundEdit.end() 才能撤消。一旦调用 CompoundEdit.end()，将撤消所有的编辑动作，即撤消当组合编辑处于进行状态时添加到组合编辑中的所有编辑动作。

图 6-8 所示的小应用程序图解说明了组合编辑的使用。这个小应用程序有一个 JList 扩展，这个扩展提供一个 undoableAdd 方法，这个方法把对象添加到列表中，然后紧接着把一个可撤销编辑发送给列表的可撤销编辑监听器。这个小应用程序还维护一个组合编辑，用它来撤销列表中的项。

左上图示出了通过六次激活 Add Item 按钮把六个选项添加到列表后的小应用程序。右上图示出了激活 End 按钮后的小应用程序，激活 End 按钮后，调用了用于组合编辑（由小应用程序维护）的 CompoundEdit.end() 方法。左下图示出了在 Undo 按钮激活后的小应用程序，Undo 按钮激活后，调用了 CompoundEdit.undo() 方法，这个方法撤销添加的所有选项。右下图示出了通过激活“Redo”按钮来调用 CompoundEdit.redo() 方法后的小应用程序。

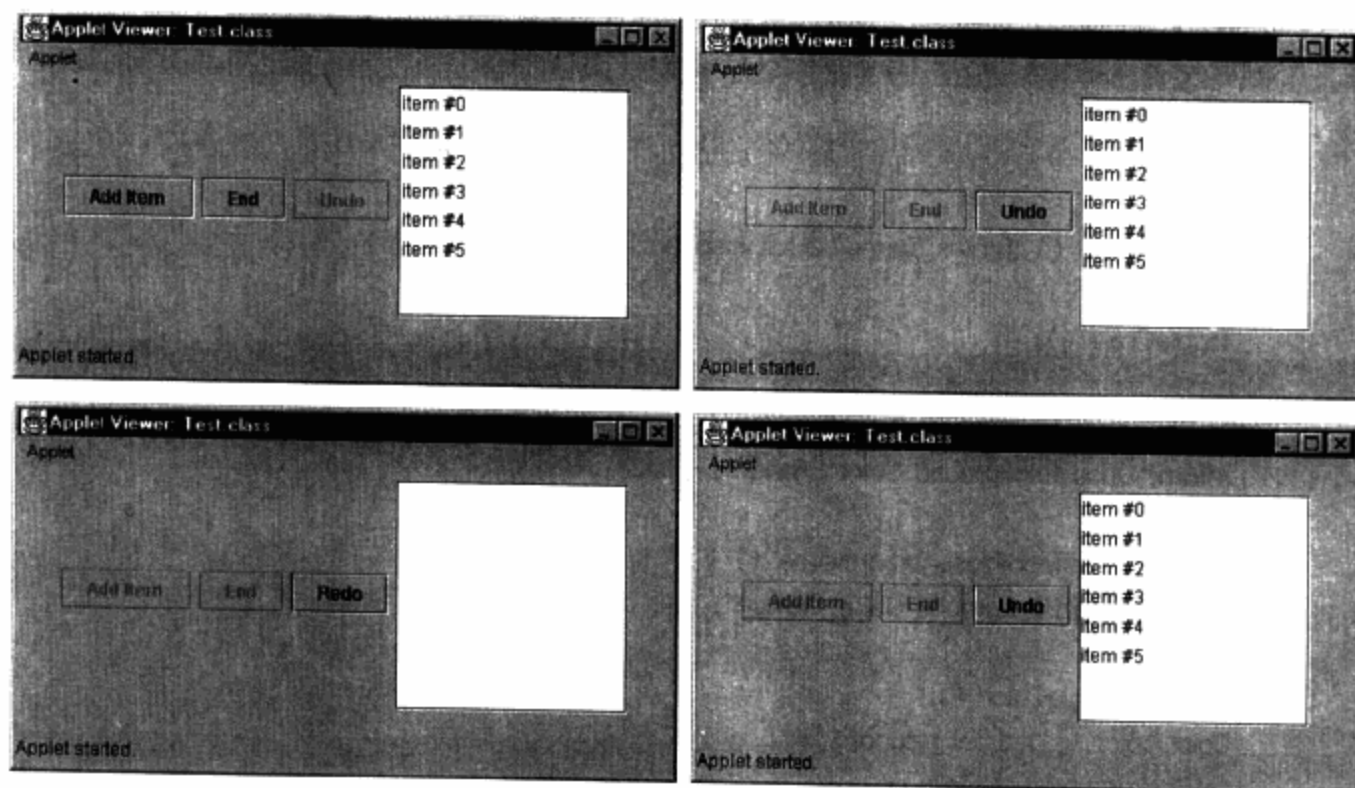


图 6-8 使用组合编辑

图 6-8 所示的小应用程序所包含的列表是一个 UndoableList 实例，它能撤销列表的项。UndoableList 类提供一个 undoableAdd 方法，这个方法把一个对象添加到列表中并激发一个 UndoableList.AddItemEdit 实例。在 UndoableEditSupport 的一个实例的援助下，将执行登记监听器和激发事件的工作。

AddItemEdit 类扩展 AbstractUndoableEdit 并通过删除添加到列表中的最后一项来实现 undo()。AddItemEdit.redo() 添加由 undo() 方法删除的那一项。

```
class UndoableList extends JList {
    UndoableEditSupport support = new UndoableEditSupport ();
    DefaultListModel model;

    public UndoableList () {
        setModel (model = new DefaultListModel ());
    }

    public void addUndoableEditListener (UndoableEditListener l) {
        support.addUndoableEditListener (l);
    }

    public void removeUndoableEditListener (
        UndoableEditListener l) {
        support.removeUndoableEditListener (l);
    }

    public void undoableAdd (Object s) {
```

```

    model.addElement (s);
    support.postEdit (new AddItemEdit ());
}
class AddItemEdit extends AbstractUndoableEdit {
    Object lastItemAdded;

    public void undo () throws CannotUndoException {
        super.undo ();
        lastItemAdded = model.getElementAt (model.getSize () -1);
        model.removeElement (lastItemAdded);
    }

    public void redo () throws CannotRedoException {
        super.redo ();
        model.addElement (lastItemAdded);
    }
}
}

```

这个小应用程序创建一个 UndoableList 实例，这个实例包裹在一个滚动窗格中，这个滚动窗格添加到小应用程序的内容窗格中。小应用程序除了有“Add Item”、“End”和“Undo”按钮外，还创建了一个 UndoAction 实例和一个 CompoundEdit 实例。

添加到 End 按钮中的动作监听器调用 CompoundEdit.end ()，添加到 Add 按钮中的动作监听器把一个字符串添加到列表中。这两个监听器都调用小应用程序的 updateButtonsEnabledState，它根据组合编辑的状态来设置这三个按钮的允许状态。

```

public class Test extends JApplet {
    private UndoableList list = new UndoableList ();
    private JScrollPane scrollPane = new JScrollPane (list);
    private JButton addButton = new JButton ("Add Item"),
        endButton = new JButton ("End"),
        undoButton = new JButton ("Undo");

    private UndoAction undoAction = new UndoAction ();
    private CompoundEdit compoundEdit = new CompoundEdit ();
    private int cnt = 0;

    public void init () {
        //add buttons and scrollpane to content pane ...

        endButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                compoundEdit.end ();
                updateButtonsEnabledState ();
            }
        });

        addButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                list.undoableAdd ("item #" + cnt++);
                updateButtonsEnabledState ();
            }
        });

        undoButton.addActionListener (undoAction);

        endButton.setEnabled (false);
        undoButton.setEnabled (false);
    }

    private void updateButtonsEnabledState () {

```

```

boolean inProgress = compoundEdit.isInProgress ();
endButton.setEnabled (inProgress);
addButton.setEnabled (inProgress);

if (undoButton.getText ().equals ("Undo"))
    undoButton.setEnabled (compoundEdit.canUndo ());
else
    undoButton.setEnabled (compoundEdit.canRedo ());
}

```

例 6-14 列出了图 6-8 所示的小应用程序的完整代码。

例 6-14 使用组合编辑

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.undo.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private UndoableList list = new UndoableList ();
    private JScrollPane scrollPane = new JScrollPane (list);

    private JButton addButton = new JButton ("Add Item"),
        endButton = new JButton ("End"),
        undoButton = new JButton ("Undo");

    private UndoAction undoAction = new UndoAction ();
    private CompoundEdit compoundEdit = new CompoundEdit ();
    private int cnt = 0;

    public void init () {
        Container contentPane = getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (addButton);
        contentPane.add (endButton);
        contentPane.add (undoButton);
        contentPane.add (scrollPane);

        scrollPane.setPreferredSize (new Dimension (150, 150));
        list.addUndoableEditListener (undoAction);

        endButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                compoundEdit.end ();
                updateButtonsEnabledState ();
            }
        });

        addButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                list.undoableAdd ("item #" + cnt++);
                updateButtonsEnabledState ();
            }
        });

        undoButton.addActionListener (undoAction);

        endButton.setEnabled (false);
        undoButton.setEnabled (false);
    }
}

```

```

    }
    private void updateButtonsEnabledState () {
        boolean inProgress = compoundEdit.isInProgress ();
        endButton.setEnabled (inProgress);
        addButton.setEnabled (inProgress);
        if (undoButton.getText ().equals ("Undo"))
            undoButton.setEnabled (compoundEdit.canUndo ());
        else
            undoButton.setEnabled (compoundEdit.canRedo ());
    }
    class UndoAction extends AbstractAction
        implements UndoableEditListener {
        public UndoAction () {
            putValue (Action.NAME, "Undo");
        }
        public void actionPerformed (ActionEvent e) {
            String name = undoButton.getText ();
            boolean isUndo = name.equals ("Undo");
            if (isUndo) compoundEdit.undo ();
            else compoundEdit.redo ();
            undoButton.setText (isUndo ? "Redo" : "Undo");
        }
        public void undoableEditHappened (UndoableEditEvent e) {
            UndoableEdit edit = e.getEdit ();
            compoundEdit.addEdit (edit);
            endButton.setEnabled (true);
        }
    }
}

class UndoableList extends JList {
    UndoableEditSupport support = new UndoableEditSupport ();
    DefaultListModel model;

    public UndoableList () {
        setModel (model = new DefaultListModel ());
    }

    public void addUndoableEditListener (UndoableEditListener l) {
        support.addUndoableEditListener (l);
    }

    public void removeUndoableEditListener (
        UndoableEditListener l) {
        support.removeUndoableEditListener (l);
    }

    public void undoableAdd (Object s) {
        model.addElement (s);
        support.postEdit (new AddItemEdit ());
    }

    class AddItemEdit extends AbstractUndoableEdit {
        Object lastItemAdded;

        public void undo () throws CannotUndoException {
            super.undo ();
            lastItemAdded = model.getElementAt (model.getSize () - 1);
        }
    }
}

```



```
model.removeElement (lastItemAdded);  
|  
public void redo () throws CannotRedoException |  
    super.redo ();  
    model.addElement (lastItemAdded);  
|
```

6.7.4 UndoManager

javax.swing.undo 提供一个扩展 CompoundEdit.UndoManager 类的 UndoManager 类。UndoManager 类在两个方面与其超类不同。首先，UndoManager 类通过把传送给它的编辑动作添加到它本身来实现 UndoableEditListener 接口。这种办法使得撤消管理器能够管理来自多个事件源的可撤消事件。

其次，撤消管理器在它进行时就可以进行撤消操作，而不像 CompoundEdit 类那样，只能在调用 CompoundEdit.end() 后才能进行撤消。在为一个 UndoManager 实例调用了 end() 后，撤消管理器有效地把多个编辑动作转换成一个组合编辑。这种特性在以下情况中是非常有用的，例如较小的编辑需要在完成前能够撤消，但是在完成后又应当作为单个编辑动作。

图 6-9 所示的小应用程序除了组合编辑由一个 UndoManager 实例所替代外，其余都与例 6-14 所列的小应用程序相同。图 6-9 所示的小应用程序没有列出来，但是本书所附光盘中可以找到这个小应用程序的代码。

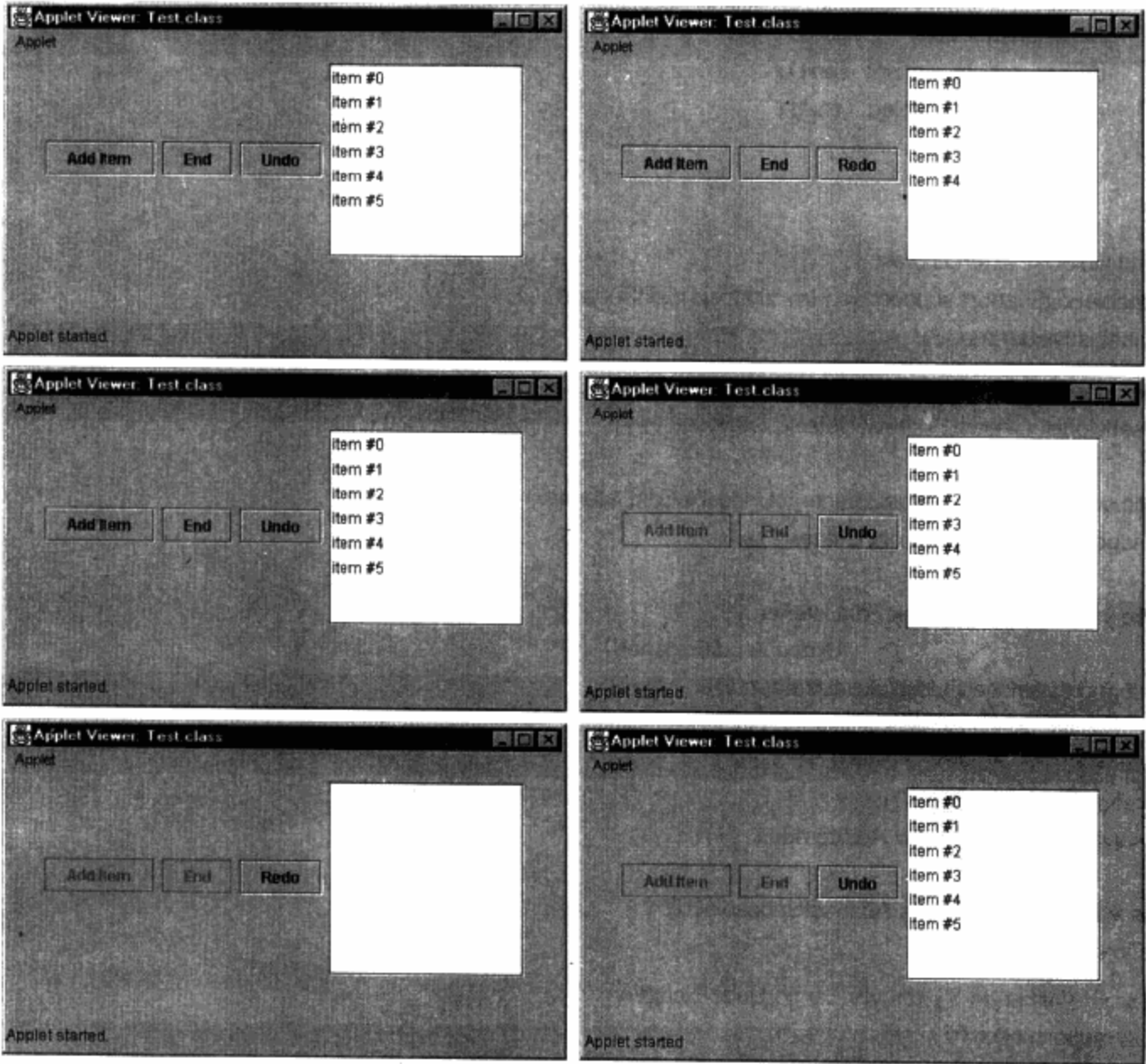


图 6-9 使用撤消管理器

图 6-9 左上图（与图 6-8 左上图一样）示出了把六个选项添加到列表中后的小应用程序。注意，图 6-9 左上图的 Undo 按钮是允许的，这与图 6-8 左上图相反，因为可以在调用管理器的 end 方法之前调用用于 UndoManager 实例的 undo 方法。图 6-9 右上图示出了在 Undo 按钮激活后的小应用程序，此时，撤消了“item # 5”。

图 6-9 中左图示出了在接着激活 Redo 按钮后的小应用程序，此时，撤消了对“item # 5”的删除。图 6-9 中右图示出了在激活 End 按钮后的小应用程序。

图 6-9 左下图示出了在接着激活 Undo 按钮后的小应用程序。因为用于撤消管理器的 end 方法已调用，现在，这个管理器的行为就像一个组合编辑，即对 UndoManager.undo 的调用将调用添加到这个管理器中的每个编辑的 undo 方法。最后，图 6-9 右下图示出了在 Redo 按钮激活后小应用程序，此时，小应用程序恢复原来添加到列表中的所有选项。

6.7.5 状态编辑

一个状态编辑可以与一个状态可编辑对象相关联，而状态可编辑对象有两个相反状态。状态编辑可以在两个状态之间切换。状态可编辑对象实现 StateEditable 接口，这个接口定义两个方法：void storeState (Hashtable) 和 void restoreState (Hashtable)。

状态编辑由 StateEdit 类来表示，它必须用一个实现 StateEditable 接口的对象来构造。当构造状态编辑且调用 StateEdit.end() 方法时，StateEdit 的实例为传送给 StateEdit 构造方法的 StateEditable 对象调用 storeState()。因此，定义了状态可编辑对象的前状态和后状态。接着，对状态编辑的 undo 和 redo 的调用将导致对 StateEditable.restoreState (对它传送了一个适当的哈希表) 的调用。

图 6-10 所示的小应用程序图解说明了状态编辑的使用。这个小应用程序包含三个按钮和一个有四个文本域的面板，这三个按钮分别是开始编辑、结束编辑和撤消或重复编辑。这个文本域包含在一个面板中，这个面板扩展 JPanel 并实现 StateEditable 接口。

左上图显示了小应用程序开始时的样子。右上图显示在激活 Start Edit 按钮后且其中的两

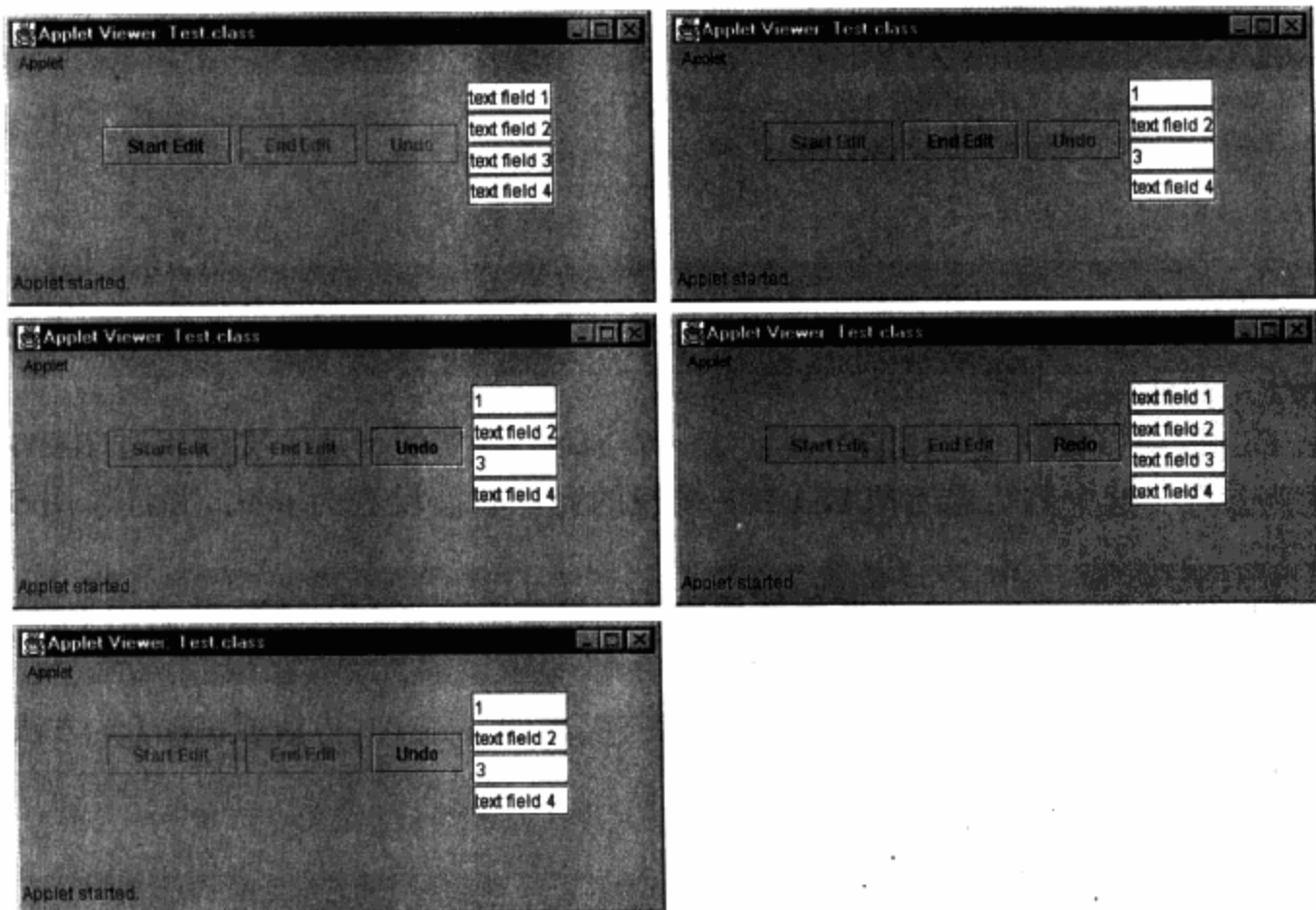


图 6-10 使用状态编辑

个文本域的内容已发生变化后小应用程序的样子。中左图显示了在 End Edit 按钮激活后的小应用程序的样子，中右图显示了在激活 Undo 按钮后的小应用程序的样子。最后，下图显示了在激活 Redo 按钮后的小应用程序的样子。

这个小应用程序所包含的面板是 TextFieldPanel 的一个实例，它包含四个由同一个 BoxLayout 实例布局的文本域。有关 BoxLayout 布局管理器的更多信息，请参见 6.5 节“BoxLayout 和 Box 类”。

TextFieldPanel.storeState 方法把对每一个文本域的引用都传送给一个哈希表，这个哈希表是作为键/值对传送给这个方法的。TextFieldPanel.restoreState 方法循环作为其参数的哈希表中的键/值对，并为以键存储在哈希表中的每一个文本域设置文本。应该注意，传送给 StateEditable.restoreState() 的哈希表只包含已改变的项。

```
class TextFieldPanel extends JPanel implements StateEditable {
    JTextField [] fields = new JTextField [] {
        new JTextField ("text field 1"),
        new JTextField ("text field 2"),
        new JTextField ("text field 3"),
        new JTextField ("text field 4"),
    };

    public TextFieldPanel () {
        setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
        for (int i = 0; i < fields.length; ++ i)
            add (fields [i]);
    }

    public void storeState (Hashtable hashtable) {
        for (int i = 0; i < fields.length; ++ i)
            hashtable.put (fields [i], fields [i].getText ());
    }

    public void restoreState (Hashtable hashtable) {
        Enumeration keys = hashtable.keys ();
        while (keys.hasMoreElements ()) {
            JTextField field = (JTextField) keys.nextElement ();
            field.setText ((String) hashtable.get (field));
        }
    }
}
```

这个小应用程序创建一个 TextFieldPanel 实例和三个按钮，接着把它们都添加到小应用程序的内容窗格中。每个按钮都有一个动作监听器。

与 Start Edit 按钮相关联的动作监听器创建一个 StateEdit 实例，这个实例以对 TextFieldPanel 的一个引用为参数。这个监听器还把 End Edit 按钮的允许状态设置为 true，并且禁用 Start Edit 按钮。

与 End Edit 按钮相关联的动作监听器调用用于编辑状态的 end() 方法并启用 Undo 按钮和禁用 End Edit 按钮。

与 Undo 按钮相关联的动作监听器调用用于状态编辑的 undo() 方法或 redo() 方法，并更新 Undo 按钮的文本。

```
public class Test extends JApplet {
    private TextFieldPanel panel = new TextFieldPanel ();
    private StateEdit stateEdit;
```

```

private JButton startButton = new JButton ("Start Edit"),
        endButton = new JButton ("End Edit"),
        undoButton = new JButton ("Undo");

public void init () {
    // add buttons and panel to content pane ...

    endButton.setEnabled (false);
    undoButton.setEnabled (false);

    startButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            stateEdit = new StateEdit (panel);
            endButton.setEnabled (true);
            startButton.setEnabled (false);
        }
    });

    endButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            stateEdit.end ();
            undoButton.setEnabled (true);
            endButton.setEnabled (false);
        }
    });

    undoButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            String name = undoButton.getText ();
            boolean isUndo = name.equals ("Undo");

            if (isUndo) stateEdit.undo ();
            else stateEdit.redo ();

            undoButton.setText (isUndo ? "Redo" : "Undo");
        }
    });
}

```

例 6-15 列出了图 6-10 所示的小应用程序的完整代码。

例 6-15 使用状态编辑

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.undo.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    private TextFieldPanel panel = new TextFieldPanel ();
    private StateEdit stateEdit;

    private JButton startButton = new JButton ("Start Edit"),
            endButton = new JButton ("End Edit"),
            undoButton = new JButton ("Undo");

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
    }
}

```

```

contentPane.add (startButton);
contentPane.add (endButton);
contentPane.add (undoButton);
contentPane.add (panel);

endButton.setEnabled (false);
undoButton.setEnabled (false);

startButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        stateEdit = new StateEdit (panel);
        endButton.setEnabled (true);
        startButton.setEnabled (false);
    }
});

endButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        stateEdit.end ();
        undoButton.setEnabled (true);
        endButton.setEnabled (false);
    }
});

undoButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        String name = undoButton.getText ();
        boolean isUndo = name.equals ("Undo");

        if (isUndo) stateEdit.undo ();
        else stateEdit.redo ();

        undoButton.setText (isUndo ? "Redo" : "Undo");
    }
});

}

class TextFieldPanel extends JPanel implements StateEditable {
    JTextField [] fields = new JTextField [] {
        new JTextField ("text field 1"),
        new JTextField ("text field 2"),
        new JTextField ("text field 3"),
        new JTextField ("text field 4"),
    };

    public TextFieldPanel () {
        setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));

        for (int i=0; i < fields.length; ++i)
            add (fields [i]);
    }

    public void storeState (Hashtable hashtable) {
        for (int i=0; i < fields.length; ++i)
            hashtable.put (fields [i], fields [i].getText ());
    }

    public void restoreState (Hashtable hashtable) {
        Enumeration keys = hashtable.keys ();

        while (keys.hasMoreElements ()) {
            JTextField field = (JTextField) keys.nextElement ();

```

```
field.setText ((String) hashtable.get (field));
```

6.8 本章回顾

Swing 提供了许多在 Swing 内部使用的实用工具。Box 和 BoxLayout 类提供了沿着水平方向或垂直方向布局组件的能力（有些能力是 AWT 所缺乏的）。SwingUtilities 类提供了许多方法，这些方法功能全面、范围广泛，已经被开发人员广泛使用。

进度监视器和支持撤消/重复是大多数用户界面的基本特性，使用也相对简单。

第 7 章 插入式界面样式

插入式界面样式是建立在第 3 章“Swing 组件体系结构”中讨论的组件体系结构基础之上的，即一个组件的界面样式是通过把一个特殊的 UI 代表插入这个组件来设置的。

Swing 还提供了一个 API 来管理界面样式。界面样式的管理包括如下几个方面：定义界面样式、指定当前的界面样式和为当前的界面样式添加附加界面样式等。例如，图 7-1 示出了一个以不同界面样式运行的小应用程序（顺时针方向，从左上开始分别是 Metal、Windows、Motif 和 Macintosh 的界面样式）。这个小应用程序提供了一个组合框，用于为该小应用程序的所有组件选取界面样式。这个小应用程序借助 Swing 的 `UIManager` 类，根据在该组合框中的选择安装界面样式^①。

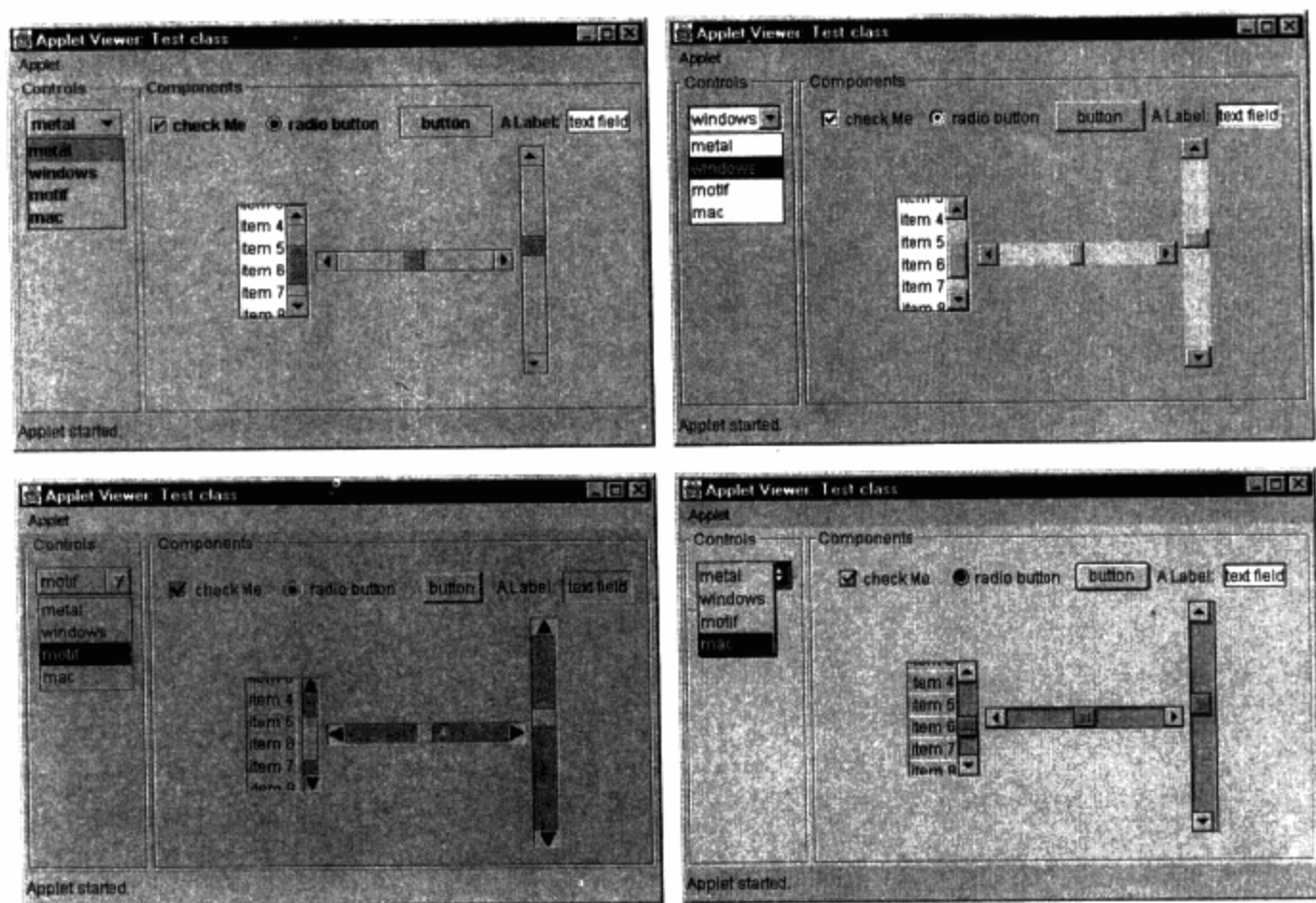


图 7-1 为多个组件改变界面样式

7.1 界面样式结构

`javax.swing` 包中的三个类：`LookAndFeel`、`UIDefaults` 和 `UIManager` 为 Swing 的界面样式提供了管理手段。

图 7-2 中示出了 `LookAndFeel`、`UIDefaults` 和 `UIManager` 类。

`LookAndFeel` 是一个抽象类。它可以扩展为一个界面样式提供一个特性。`LookAndFeel` 类提供了很多方便的 `static` 方法，还定义了抽象方法来指定如下界面样式属性：

① 未列出图 7-1 中示出的小应用程序的代码；例 7-3 给出了一个改变界面样式的例子。

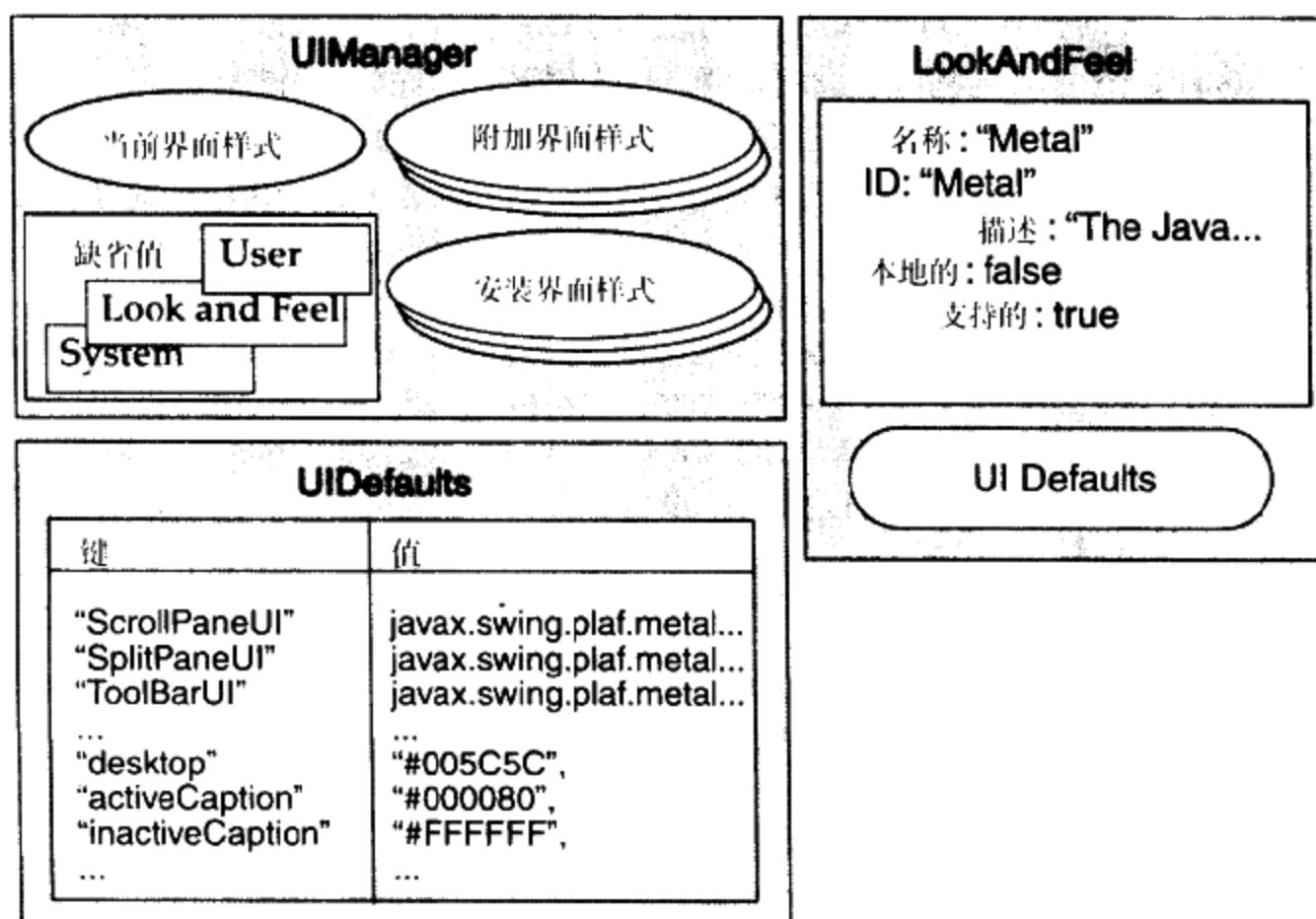


图 7-2 界面样式结构

- 名称。
- ID。
- 描述。
- 本地的。
- 支持的。

上述界面样式属性在本章后面讨论。

UIDefaults 是 `java.util.Hashtable` 的一个直接扩展，它维护界面样式的缺省值。开发人员可以不直接操纵 UIDefaults 实例，而通过 UIManager 类访问缺省值。

UIManager 类是一个目录服务，为下述操作提供了 static 访问方法：

- 安装界面样式。
- 当前界面样式。
- 当前界面样式的缺省值。
- 附加界面样式。
- 系统的类名和跨平台的界面样式。

UIManager 类实现的所有方法都是静态的。

7.1.1 界面样式

人们常问的一个问题是：Swing 是否支持显示多行文本的标签？答案是即支持又不支持。Swing 的标签是 JLabel 的实例，而 JLabel 并不对显示多行文本提供支持，从这方面来说，答案是否定的。

另一方面，Swing 包含 JTextArea 和 JEditPane 等组件，这些组件在显示多行文本方面是很在行的。把一个文本组件（比如，JTextArea 的一个实例）所提供的功能用于多行标签组件的途径似乎是存在的。用一个文本域来实现一个多行文本的问题是，一个文本域感觉起来像一个文本域，而不像一个标签。

Swing 的可插入界面样式结构允许用一个组件的缺省特性，如边框、颜色和字体等来设置另一个组件。这样，一个文本域可以设置成像一个标签——这就是关键所在。

界面样式用保存在一个 `UIDefaults` 实例（一个哈希表）中的已知名称定义一组缺省属性。参见附录 B “插入式界面样式常数”，其中完整地列出了已知属性名和相应的对象类型。

`LookAndFeel` 类为在一个 `JComponent` 实例上安装缺省属性提供了方便的方法：

```
Public static void installBorder (JComponent, String border)
```

```
Public static void installColors (JComponent, String foreground, String background)
```

```
Public static void installColorsAndFont (JComponent, String foreground, String background, String font)
```

例如，`installBorder` 方法根据传送给它的字符串参数（一个已知名的属性）设置一个组件的边框。例如，一个按钮的缺省边框可以这样来设置：

```
// code fragment: installing a button's default border
// "Button.border" is a well-known name for default button border
JButton button = new JButton ("a button");
LookAndFeel.installBorder (button, "Button.border");
```

因此，采用如下方式，可以使一个文本域具有按钮的缺省边框：

```
// code fragment
JTextArea textArea = new JTextArea ();
LookAndFeel.installBorder (textArea, "Label.border");
```

图 7-3 中示出的小应用程序包含了一个标签作为这个小应用程序内容中的 North 组件，并包含了 `MultilineLabel` 的一个实例作为中间组件。图 7-3 示出了改变这个小应用程序的情况，以说明多行标签的功能。

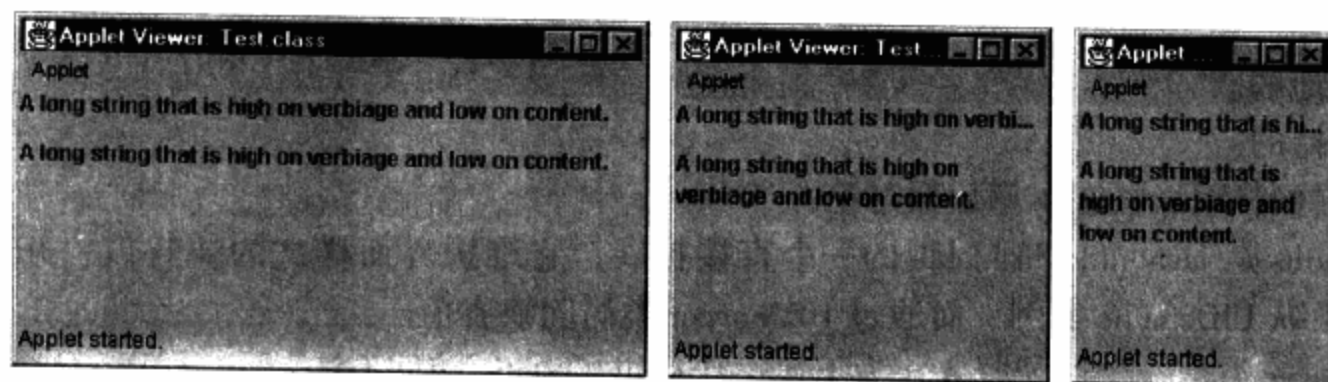


图 7-3 多行标签

例 7-1 中列出了图 7-3 中示出的小应用程序的代码。

例 7-1 一个多行标签

```
import javax.swing.* ;
import javax.swing.plaf.BorderUIResource;
import java.awt.* ;
import java.awt.event.* ;
import java.util.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JLabel label = new JLabel (
            "A long string that is high on verbiage and " +
            "low on content.");
```

```

MultilineLabel multilineLabel = new MultilineLabel (
    "A long string that is high on verbiage and " +
    "low on content.");

contentPane.setLayout (new BorderLayout (2, 10));

contentPane.add (label, BorderLayout.NORTH);

contentPane.add (multilineLabel, BorderLayout.CENTER);
}

}

class MultilineLabel extends JTextArea {
public MultilineLabel (String s) {
    super (s);
}

public void updateUI () {
    super.updateUI ();

    // turn on wrapping and disable editing and highlighting

    setLineWrap (true);
    setWrapStyleWord (true);
    setHighlighter (null);
    setEditable (false);

    // Set the text area's border, colors and font to
    // that of a label

    LookAndFeel.installBorder (this, "Label.border");
    LookAndFeel.installColorsAndFont (this,
        "Label.background",
        "Label.foreground",
        "Label.font");
}
}

```

这个小应用程序的 `init` 方法创建了一个 `JLabel` 实例和一个添加到这个小应用程序内容窗格中的 `MultilineLabel` 实例。

`MultilineLabel` 类扩展 `JTextArea`，并重载 `updateUI` 方法，以便把一个文本域伪装成一个标签。重复一下，一个组件的 `updateUI` 方法根据当前界面样式更新该组件的 UI 代表。扩展 `updateUI` 确保界面样式每次变化时都对文本域进行伪装。

这个文本域的 `lineWrap` 属性设置为 `true`，`wrapStyleWord` 属性也是如此。因为标签不能加重显示或编辑，因此，这个文本域的 `highlighter` 属性设置为 `null`，并且可编辑性属性设置为 `false`。

通过调用 `LookAndFeel.installBorder()` 和 `LookAndFeel.installColorsAndFont()` 来设置这个文本域的边框、颜色和字体。边框、颜色和字体是用 `JLabel` 类的已知名的属性名来指定的。

1. LookAndFeel 类

`LookAndFeel` 类提供了一些方便的 `static` 方法，其中 `installBorder` 和 `installColorsAndFont` 两个方法在前面已作过介绍。`LookAndFeel` 类还定义了很多抽象方法来个性化设置一个界面样式。

```
public abstract String getName ()
```

```

public abstract String getDescription ()
public abstract String getID ()

public abstract boolean isNativeLookAndFeel ()
public abstract boolean isSupportedLookAndFeel ()

```

一个界面样式的名称应当是一个短名称，如 Windows 或 Metal。通常用 GetName() 方法返回的字符串在菜单和列表中标识界面样式。

getDescription 方法应当返回对界面样式的简短描述。描述一般是一两个描述这个名字的句子；例如，描述可能包含一个版权信息。

getID 方法返回一个用于标识界面样式的字符串。Swing 没有使用 getID 方法，它也没有说明应该如何使用一个界面样式的 ID。但是，一般来说，从 getID 返回的字符串应当返回一个标识界面样式的唯一字符串。

如果一个界面样式，仿真一个小应用程序或应用程序所运行的平台的界面样式，则这个界面样式是本地的。如果一个界面样式不仿真一个平台特有的界面样式，则 isNativeLookAndFeel 应当返回 false。为了确定一个界面样式是否仿真本地平台，可以从系统属性中取得操作系统名。例如，Windows 的界面样式以如下方式实现 isNativeLookAndFeel：

```

// From javax.swing.plaf.Windows
public boolean isNativeLookAndFeel () {
    String osName = System.getProperty("os.name");
    return (osName != null) && (osName.indexOf("Windows") != -1);
}

```

如果操作系统名包含 Windows，则 WindowsLookAndFeel.isNativeLookAndFeel() 方法返回 true；否则该方法返回 false。

如果 isNativeLookAndFeel() 返回 true，则这个界面样式就是被支持的。可能会有法规的或竞争方面的原因禁止在某个平台上使用一个特定的界面样式。例如，Windows 只支持 Windows 类型的界面样式。WindowsLookAndFeel 类以如下方式实现 isSupportedLookAndFeel()：

```

// From javax.swing.plaf.Windows
public boolean isSupportedLookAndFeel () {
    return isNativeLookAndFeel ();
}

```

每个界面样式都实现一个最终扩展 javax.swing.LookAndFeel 的类。图 7-4 示出了 Swing 中界面样式类的层次结构。

BasicLookAndFeel 和 MultiLookAndFeel 扩展 LookAndFeel 类^①。

BasicLookAndFeel 类定义了大多数界面样式类

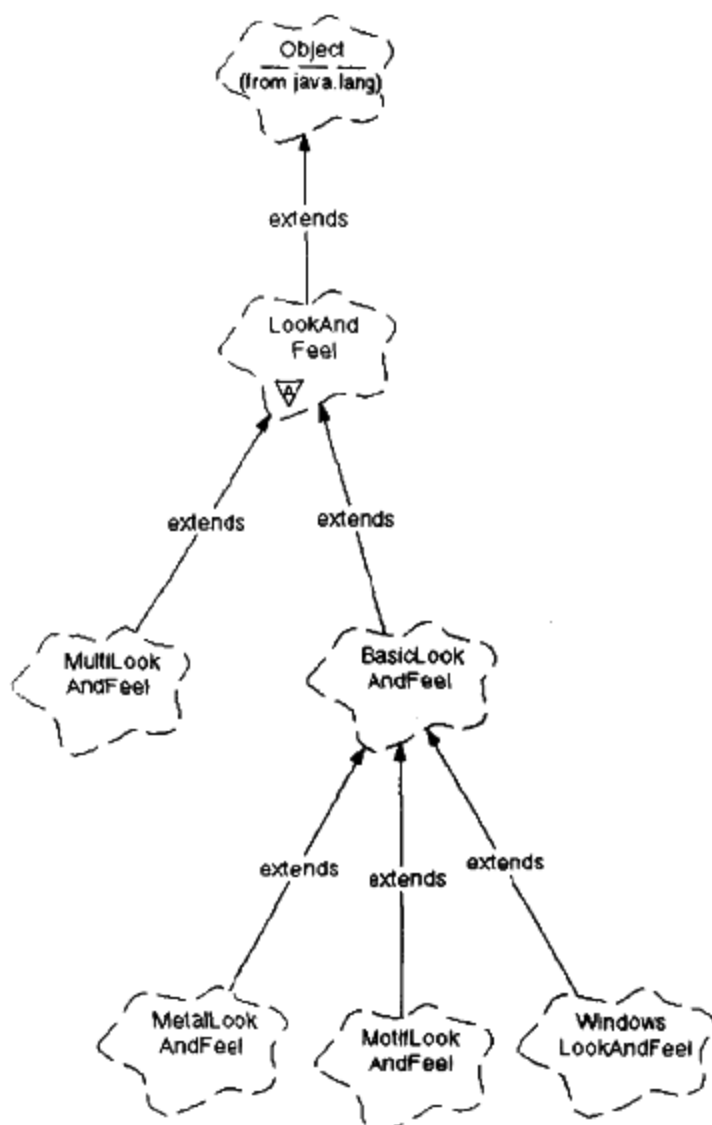


图 7-4 LookAndFeel 类的层次结构

① 与 MultiLookAndFeel 类有关的更多信息参见 7.3 节“附加 UI”。

都具有的缺省属性；例如，BasicLookAndFeel 定义一个缺省表，其中包含如下按钮属性：

```
// From BasicLookAndFeel.java
Object [] defaults = {
    // * * * Label
    "Label.font", dialogPlain12,
    "Label.background", table.get("control"),
    "Label.foreground", table.get("controlText"),
    "Label.disabledForeground", white,
    "Label.border", null,

    // Default properties for other components follow
    |
}
```

这些属性存储在一个哈希表中，能够被 BasicLookAndFeel 的扩展修改。

类总结 7-1 中对 LookAndFeel 类作了总结。

类总结 7-1 LookAndFeel

扩展：java.lang.Object

1. 构造方法

```
public LookAndFeel ()
```

这个 LookAndFeel 的无参数构造方法是由编译器产生的，因为 LookAndFeel 类不实现它自己的构造方法。

2. 方法

(1) 初始化/toString ()

```
public void initialize ()
public void uninitialize ()
public String toString ()
```

当安装一个小应用程序或应用程序的初始界面样式时，调用 initialize 方法，而当替换初始界面样式时则调用 uninitialize 方法。LookAndFeel 类为这些方法提供空实现。

LookAndFeel 类重载 toString 以返回一个字符串，这个字符串包含这个界面样式的描述，在此描述后面还跟有其类名。

(2) 安装边框、颜色和字体

```
public static void installBorder (JComponent, String defaultBorderName)
public static void installColors (JComponent, String defaultBackgroundName,
                                   String defaultForegroundName)
public static void installColorsAndFont (JComponent, String defaultBackgroundName,
                                          String defaultForegroundName, String defaultFontName)
public static void uninstallBorder (JComponent)
```

上述方法为传送给上述安装方法的组件安装那些已有名称的缺省属性。第 7.1.1 节“界面样式”中介绍了其中的两种方法：installBorder () 和 installColorsAndFont ()。

这三个方法都从 UIManager 获得相应的缺省属性，这些缺省属性也就用来设置组件。例如，LookAndFeel.installBorder () 是以如下方式实现的：

```
// From LookAndFeel.java
public static void installBorder (JComponent c,
    Border b = c.getBorder ();
    if (b == null || b instanceof UIResource) |
```



```
c.setBorder (UIManager.getBorder (defaultBorderName));
```

仅当当前属性是 null 或一个 UI 资源时才设置这个属性。Swing 区分应用和 UI 资源，仅重新设置那些标记为 UI 资源的值。有关 UI 资源的更多信息参见 7.1.4 节“UI 资源”。

(3) 缺省值

```
public UIDefaults getDefaults ()
```

对一个给定的界面样式调用 getDefaults 方法。它返回一个 UIDefaults 实例，这个实例包含了这个界面样式的缺省属性。有关 UIDefaults 类的更多情况参见 7.1.2 节。

(4) 描述属性

```
public abstract String getDescription ()
```

```
public abstract String getID ()
```

```
public abstract String getName ()
```

```
public abstract String isNativeLookAndFeel ()
```

```
public abstract String isSupportedLookAndFeel ()
```

上面都是用来定制界面样式的抽象方法。需要 LookAndFeel 的扩展来提供实现方法。

上面列出的抽象方法在前面 7.1.1 节“界面样式”节中进行过较深入的讨论。

(5) 方便的静态方法

```
public static Object makeIcon (Class, String)
```

```
public static JtextComponent.KeyBinding [] makeKeyBindings (Object [])
```

上面列出了扩展 LookAndFeel 类的方便方法。makeIcon 方法返回一个闲置值，这个值在需要时创建一个图像。有关闲置值的更多信息参见 7.1.2 节。makeKeyBindings 方法创建 Swing 文本组件的键值绑定。

7.1.2 界面样式缺省值

所有的界面样式都维护一个包含缺省组件属性的哈希表；例如，BasicLookAndFeel 为按钮边框、背景、前景颜色和图标等定义了缺省值。

```
//From javax.swing.plaf.basic.BasicLookAndFeel
```

```
Object [] defaults = {
```

```
    // * * * Buttons
```

```
    "Button.font", dialogPlain12,
```

```
    "Button.background", table.get ("control"),
```

```
    "Button.foreground", table.get ("controText"),
```

```
    "Button.border", buttonBorder,
```

```
    "Button.margin", new InsetsUIResource (2, 14, 2, 14),
```

```
    "Button.textIconGap", new Integer (4),
```

```
    "Button.textShiftOffset", new Integer (4),
```

```
    "ToggleButton.font", dialogPlain12,
```

```
    "ToggleButton.background", table.get ("control"),
```

```
    "ToggleButton.foreground", table.get ("controText"),
```

```
    "ToggleButton.border", buttonToggleButtonBorder,
```

```

"ToggleButton.margin", new InsetsUIResource (2, 14, 2, 14),
"ToggleButton.textIconGap", new Integer (4),
"ToggleButton.textShiftOffset", new Integer (0),

// many more defaults follow...

```

```
};
```

上面列出的缺省值数组定义了用 `Button.background` 和 `Button.foreground` 等名字标识的缺省值。上面列出的数组已经大量删减了；如果在本书中把整个数组都列出的话，其清单将占用9页纸。

界面样式的缺省值是在 `UIDefaults` 的一个实例中维护的；但是，开发人员最好不要直接访问这些缺省值。可以用 `UIManager` 类来访问当前界面样式的缺省值。有关 `UIManager` 类的更多情况参见 7.1.3 节“UI 管理器”小节。

UI 代表用界面样式的缺省值来设置一个组件的属性。例如，`BasicButtonUI` 类在 `BasicButtonUI.installDefaults` 方法中为一个按钮安装缺省值。

```

// From javax.swing.plaf.basic.BasicButtonUI;
private final static String propertyPrefix = "Button" + ".";
...
protected String getPropertyPrefix () {
    return propertyPrefix;
}
...
protected void installDefaults (AbstractButton b) {
    String pp = getPropertyPrefix ();
    if (! defaults-initialized) {
        defaultTextIconGap =
            ((Integer) UIManager.get (pp + "textIconGap")).intValue ();
        defaultTextShiftOffset =
            ((Integer) UIManager.get (pp +
                                     "textShiftOffset")).intValue ();
        defaults-initialized = true;
    }
    ...
    if (b.getMargin () == null ||
        (b.getMargin () instanceof UIResource)) {
        b.setMargin (UIManager.getInsets (pp + "margin"));
    }
    LookAndFeel.installColorsAndFont (b, pp +
        "background", pp + "foreground", pp + "font");
    LookAndFeel.installBorder (b, pp + "border");
}

```

`BasicButtonUI.installDefaults` 方法用 `UIManager` 的静态方法为当前界面样式中的按钮获得缺省值。`LookAndFeel` 的静态方法用来安装缺省边框、颜色和字体。`LookAndFeel` 的另一种用法参见图 7-3 “多行标签”。

图 7-5 中示出了两个几乎一样的小应用程序。每个小应用程序都用 `JTree` 的无参数构造方

法创建一个树，并把该树添加到这个小应用程序的内容窗格中。左边小应用程序显示的是一个 Java 的缺省界面样式树，而右边则是修改一些缺省值后的结果。

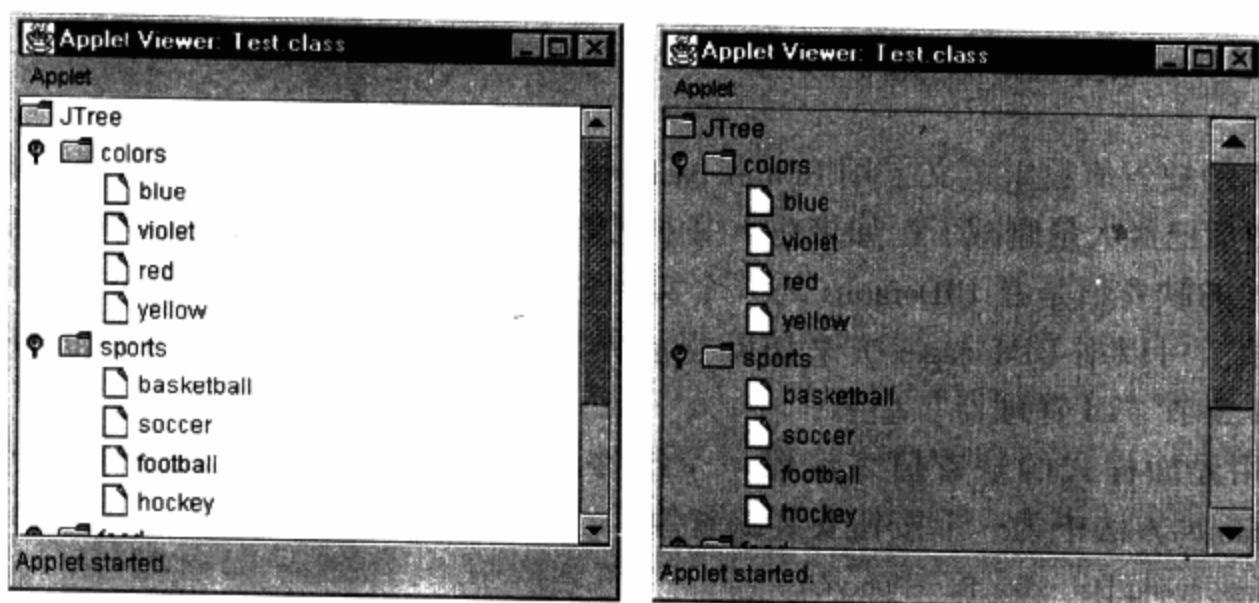


图 7-5 修改 UI 缺省值

例 7-2 列出了图 7-5 中示出的小应用程序的源代码。

例 7-2 修改 UI 缺省值

```
import java.awt.Color;
import javax.swing.*;

public class Test extends JApplet {
    public void init () {
        UIManager.put ("Tree.background", Color.lightGray);
        UIManager.put ("Tree.textBackground", Color.lightGray);

        //ScrollBar.width is peculiar to Metal L&F
        UIManager.put ("ScrollBar.width", new Integer (25));

        getContentPane ().add (new JScrollPane (new JTree ()));
    }
}
```

这个小应用程序简单地用 `UIManager.put` 方法把缺省的树背景和文本背景色改为灰色，并把滚动条的缺省宽度改为 25 像素。

滚动条的缺省宽度是通过改变滚动条的缺省值来修改的，而不是改变树的缺省值来修改。因此，Swing 的轻量组件使用的所有垂直滚动条将具有 25 个像素的宽度。

1. UIDefaults 类

`UIDefaults` 是一个维护界面样式的缺省值的哈希表，并在这些缺省值改变时产生属性变化事件。`UIDefaults` 类还提供了方便的方法来减少对从哈希表中获取的值进行类型转换。这些值是以 `Objects` 类型存储在哈希表中的。

类总结 7-2 总结了 `UIDefaults` 类。

类总结 7-2 UIDefaults

扩展：`java.util.Hashtable`

1. 构造方法

```
public UIDefaults ()
public UIDefaults (Object [] keyValueArray)
```

UIDefaults 可以用一个对象/键值数组来创建。这个数组交替地包含键值和对象。例如，传送给 UIDefaults 构造方法的数组可能与下述内容类似：

```
... "Label.background", Color.Red, "Label.foreground", Color.Blue, ...
```

2. 方法

(1) 属性变化事件

```
public synchronized void addPropertyChangeListener (PropertyChangeListener)
public synchronized void removePropertyChangeListener (PropertyChangeListener)

protected void firePropertyChange (String propertyName,
    * Object oldValue,
    Object newValue)
```

UIDefaults 的实例区别它们的 Hashtable 超类的方式之一是：在缺省值改变时，UIDefaults 实例激发属性变化事件。UIDefaults 类为属性变化监听器提供公共登记方法，并提供了一个改变属性的私有方法。

(2) 属性插入方法

```
public Object put (Object key, Object value)
public Object get (Object)
public void putDefaults (Object [])
```

这里重载了 Hashtable 中的 put 方法，以便在关键字是一个字符串时激发一个属性变化事件。put 方法返回一个旧值。还重载了 get 方法，以便处理存储在一个哈希表中的对象，这些对象是闲置的或活动的。有关活动值和闲置值的更多信息参见下一小节“活动值与闲置值”。

putDefaults 方法把传送给它的 Object 数组中的所有关键字和值都放入哈希表。

(3) 属性访问方法

```
public Border getBorder (Object key)
public Color getColor (Object key)
public Dimension getDimension (Object key)
public Font getFont (Object key)
public Icon getIcon (Object key)
public Insets getInsets (Object key)
public int getInt (Object key)
public String getString (Object key)

public ComponentUI getUI (JComponent)
public Class getUIClass (String)
public Class getUIClass (String, ClassLoader)
protected void getUIError (String)
```

上面列出的第一组方法是访问某些属性的方法，可以用来减少类型转换。例如，一个键值为“Label.border”的类可以用以下方式从一个 UIDefaults 实例获得边框属性：

```
// code fragment
UIDefaults uiDefaults = ...
```

```
Border b = (Border) uiDefaults.get("Label.border");
```

LookAndFeel.getBorder 方法允许在不进行类型转换的情况下获得边框属性：

```
// code fragment
```

```
UIDefaults uiDefaults = ...
```

```
Border b = uiDefaults.getBorder("Label.border");
```

第二组方法在给定一个组件的情况下创建 UI 代表。有关创建 UI 代表请参见 4.2.6 节。

2. 活性值与惰性值

一些缺省属性，如内部窗体边框或复选框菜单项图标等，是很少访问的。在这种情况下，把这些缺省值的创建工作推迟到首次访问它们时是有好处的。

界面样式通常把不经常访问的值当作惰性值^①并存储在一个缺省表中。惰性值是一个实现 UIDefaults.LazyValue 接口的对象，该接口定义了唯一的一个方法：

```
Object createValue (UIDefaults table)
```

UIDefaults.get 方法在返回一个值之前，检查该值是否是 UIDefaults.LazyValue 的一个实例；如果是，则通过调用这个惰性值的 createValue 方法来获得真实值。在得到真实值之后，UIDefaults.get 方法用缺省表中的真实值替换惰性值，并返回真实值。

UIDefaults 类为那些每次访问时必须创建的值定义为另一个接口——UIDefaults.ActiveValue。除了在缺省表中活性值不被真实值替代外，UIDefaults.get() 对待活性值的方式与对待惰性值的方式相同，因此，每次访问时创建活性值。

7.1.3 UI 管理器

顾名思义，UIManager 类管理信息，即管理 Swing 小应用程序和应用程序界面样式的状态。

UIManager 类为访问提供了如下信息和服务的静态方法：

- 设置 Swing 小应用程序或应用程序的界面样式。
- 提供对当前界面样式的缺省值的访问。
- 提供对界面样式状态信息的访问。
- 通知界面样式的变化情况。
- 从 Swing 属性文件中装入 Swing 属性。

为 Swing 小应用程序或应用程序设置界面样式的能力是 UIManager 最显著的特性，该特点将在下面“设置界面样式”小节中讨论。

UIManager 提供的界面样式状态信息包括当前的界面样式、已安装的界面样式、附加界面样式等。UIManager 还管理三组缺省值：用户、界面样式和系统，参见下面的“访问 UI 缺省值”小节。

UIManager 为登记属性变化监听器提供了一些 static 方法。属性变化监听器在安装一种界面样式后会得到通知。最后，当 UIManager 初始化时，通常是当通过 UIManager.getUI() 创建第一个 UI 代表时，UIManager 从 Swing 属性文件中装载 Swing 属性。

1. 访问 UI 缺省值

UIManager 用 static put (Object key, Object value) 和 get (Object key) 方法来管理对 UIDefaults 的访问。UIManager 在幕后维护用户、界面样式和系统缺省值。

① 创建代价高的缺省值适合于作为惰性值。

对 `UIManager.put (Object key, Object value)` 的一个调用把键/值对放入用户缺省值，而对 `UIManager.get (Object key)` 的一个调用则以用户、界面样式和系统缺省值的顺序搜索，并返回搜索到的第一个匹配值。这意味着，用户缺省值的优先级高于界面样式缺省值，而界面样式缺省值的优先级又高于系统缺省值。

2. 访问界面样式

`UIManager` 类还为访问如下界面样式信息提供了 `static` 方法：

- 当前的界面样式。
- 当前界面样式的缺省值。
- 已安装的界面样式。
- 附加界面样式。

附加界面样式在 7.3 节“附加 UI”中讨论。

3. 设置界面样式

图 7-6 示出的小应用程序提供了改变其界面样式的按钮。这个界面样式是用 `UIManager.setLookAndFeel` 方法来设置的。

下面列出了一个动作监听器的代码，这个监听器处理这个小应用程序中单选钮的激活动作。它根据选取了哪个按钮来设置相应的界面样式。

`UIManager.setLookAndFeel()` 设置当前的界面样式，但不更新任何已有的组件的 UI 代表。`SwingUtilities.updateComponentTreeUI()` 递归地为包含在这个方法的容器中的所有组件更新 UI 代表。

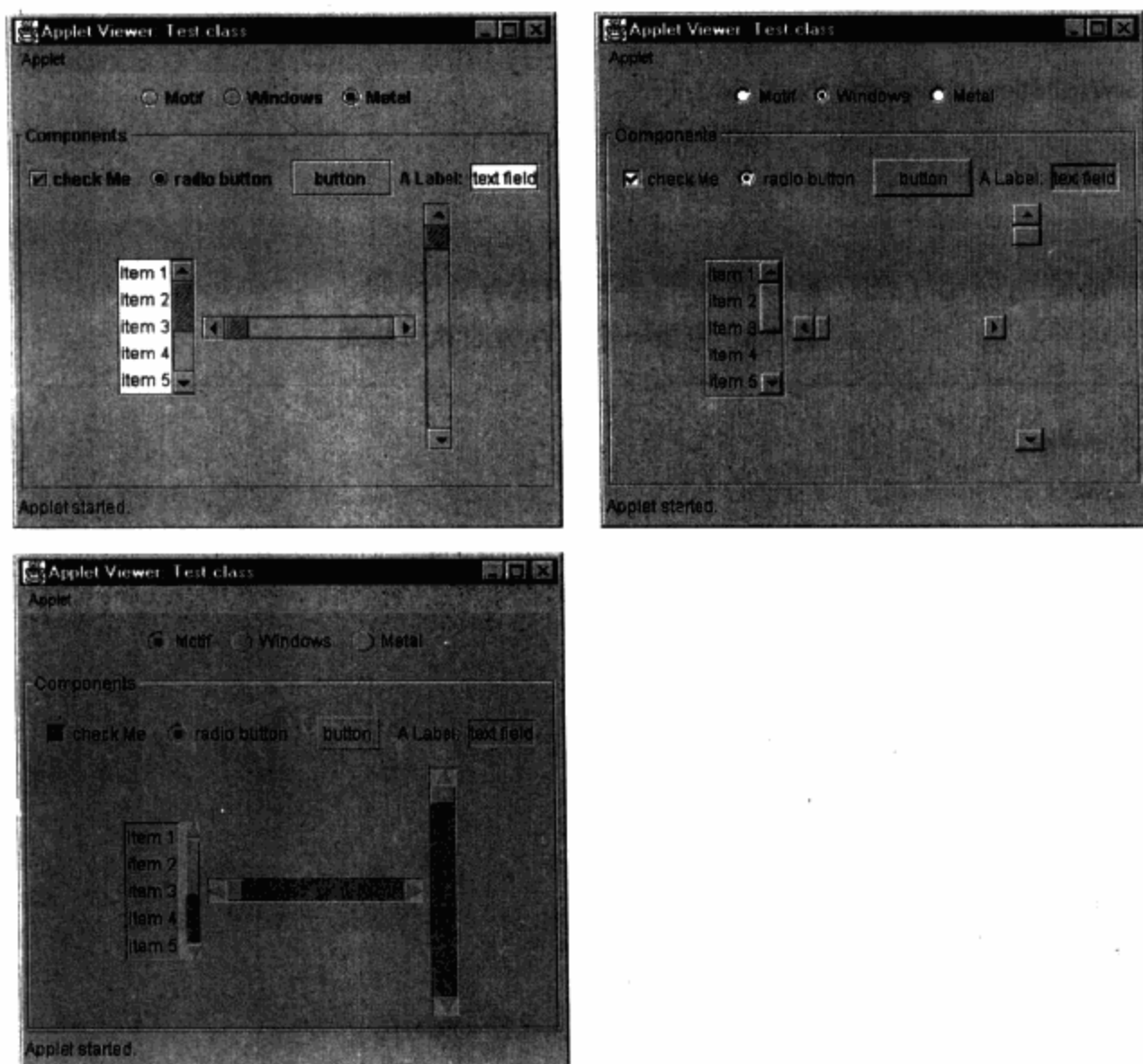


图 7-6 为多个组件设置界面样式


```
class RadioHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JRadioButton src = (JRadioButton) e.getSource ();

        try {
            if (src == motifButton)
                UIManager.setLookAndFeel (
                    "java.swing.plaf." +
                    "motif.MotifLookAndFeel");

            else if (src == windowsButton)
                UIManager.setLookAndFeel (
                    "javax.swing.plaf." +
                    "windows.WindowsLookAndFeel");

            else if (src == metalButton)
                UIManager.setLookAndFeel (
                    "javax.swing.plaf.metal." +
                    "MetalLookAndFeel");

        }
        catch (Exception ex) {
            ex.printStackTrace ();
        }

        SwingUtilities.updateComponentTreeUI (
            getContentPane ());
    }
}
```

例 7-3 中完整地列出了图 7-6 中示出的小应用程序的代码。

例 7-3 变换一个组件树的界面样式

```
import javax.swing. * ;
import java.awt. * ;
import java.awt.event. * ;
import java.util. * ;

import javax.swing.plaf.motif.MotifLookAndFeel;
import javax.swing.plaf.windows.WindowsLookAndFeel;
import javax.swing.plaf.metal.MetalLookAndFeel;
import javax.swing.plaf.ColorUIResource;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        contentPane.add (new ControlPanel (), BorderLayout.NORTH);
        contentPane.add (new ComponentPanel (),
            BorderLayout.CENTER);
    }
}
```

```

|
class ComponentPanel extends JPanel {
    public ComponentPanel () {
        JList list;
        JScrollBar sb;

        setBorder (
            BorderFactory.createTitledBorder ("Components"));

        add (new JCheckBox ("check Me"));
        add (new JRadioButton ("radio button"));
        add (new JButton ("button"));
        add (new JLabel ("A Label:"));
        add (new JTextField ("text field"));
        add (new JScrollPane (list = new JList (new Object [] {
            "item 1", "item 2", "item 3",
            "item 4", "item 5", "item 6",
            "item 7", "item 8", "item 9",
            })));

        add (sb = new JScrollBar (SwingConstants.HORIZONTAL));
        sb.setPreferredSize (new Dimension (150, 17));

        add (sb = new JScrollBar (SwingConstants.VERTICAL));
        sb.setPreferredSize (new Dimension (20, 175));

        list.setVisibleRowCount (5);
    }
}

|
class ControlPanel extends JPanel {
    JCheckBox checkBox = new JCheckBox ("UIResource");
    JRadioButton motifButton = new JRadioButton ("Motif"),
        windowsButton = new JRadioButton ("Windows"),
        metalButton = new JRadioButton ("Metal");

    public ControlPanel () {
        ActionListener listener = new RadioHandler ();
        ButtonGroup group = new ButtonGroup ();

        group.add (motifButton);
        group.add (windowsButton);
        group.add (metalButton);

        motifButton.addActionListener (listener);
        windowsButton.addActionListener (listener);
        metalButton.addActionListener (listener);

        add (motifButton);
        add (windowsButton);
        add (metalButton);
    }
}

|
class RadioHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JRadioButton src = (JRadioButton) e.getSource ();

        try {
            if (src == motifButton)
                UIManager.setLookAndFeel (

```

```

        "java.swing.plaf." +
        "motif.MotifLookAndFeel");

    else if (src == windowsButton)
        UIManager.setLookAndFeel (
            "java.swing.plaf." +
            "windows.WindowsLookAndFeel");

    else if (src == metalButton)
        UIManager.setLookAndFeel (
            "javax.swing.plaf.metal." +
            "MetalLookAndFeel");
    }
    catch (Exception ex) {
        ex.printStackTrace ();
    }
    SwingUtilities.updateComponentTreeUI (
        getContentPane ());
}

```

类 7-3 对 UIManager 类进行了总结。

类总结 7-3 UIManager

扩展: java.lang.Object

实现: java.io.Serializable

1. 构造方法

```
public UIManager ()
```

UIManager 的这个构造方法是由编译器产生的。

2. 方法

(1) 创建方法

(2) 属性变化监听器

```
public static synchronized void addPropertyChangeListener (PropertyChangeListener)
```

```
public static synchronized void removePropertyChangeListener (PropertyChangeListener)
```

UIManager.setLookAndFeel() 向所有用上面列出的 addPropertyChangeListener 方法登记属性变化监听器激发一个属性变化事件。可调用 removePropertyChangeListener 删除属性变化监听器。

注意, 上面列出的方法是 static 方法。通常, 监听器是向一个对象登记的, 但上面列出的 static 方法实际上是向 UIManager 类进行登记。

(3) 缺省值

```
public static Object put (Object key, Object value)
```

```
public static Object get (Object key)
```

```
public static Border getBorder (Object)
```

```
public static Color getColor (Object)
```

```
public static UIDefaults getDefaults (Object)
```

```
public static Dimension getDimension (Object)
```

```
public static Font getFont (Object)
```

```
public static Icon getIcon (Object)
```

```

public static Insets getInsets (Object)
public static int getInt (Object)
public static String getString (Object)
public static ComponentUI getUI (Object)

```

上面列出的所有方法都是传递当前界面样式的缺省值的方法。例如，`UIManager.put()`得到对用户缺省值的引用，然后调用 `UIDefaults.put()` 方法。

`UIManager.get()` 也是一个参数传递方法，它将参数值传递给 `UIDefaults.get()`；但是，其返回值来源于用户缺省值、界面样式缺省值或系统缺省值。对 `UIManager` 返回缺省值的方式的更多讨论参见 7.1.3 节“UI 管理器”。

(4) 界面样式

```

public static UIManager.LookAndFeelInfo [] getInstalledLookAndFeels ()
public static void setInstalledLookAndFeels (UIManager.LookAndFeelInfo []) throws SecurityException
public static void addAuxiliaryLookAndFeel (LookAndFeel)
public static LookAndFeel [] getAuxiliaryLookAndFeels ()
public static boolean removeAuxiliaryLookAndFeel (LookAndFeel)

public static String getCrossPlatformLookAndFeelClassName ()
public static String getSystemLookAndFeelClassName ()

public static void installLookAndFeel (String, String)
public static void installLookAndFeel (UIManager.LookAndFeelInfo)

public static void setLookAndFeel (String) throws ClassNotFoundException,
    InstantiationException, IllegalAccessException,
    UnsupportedLookAndFeelException

    public static void setLookAndFeel (LookAndFeel) throws
        UnsupportedLookAndFeelException

```

上面列出的所有方法皆与界面样式有关。`Swing` 跟踪记录已安装的界面样式、当前界面样式和附加界面样式。上面列出的这些 `static UIManager` 方法不仅返回系统和跨平台的界面样式的名称，它们还能访问 `Swing` 的界面样式。

7.1.4 UI 资源

如果一个按钮的前景色已明确地设置了，且这个按钮的 UI 代表随后改变了^①，那么这个按钮的前景色改变为新的界面样式的按钮的缺省前景色吗？这个问题的答案取决于该颜色是作为一个 UI 资源设置的，还是作为一种颜色设置的。若是作为一个 UI 资源设置的，则答案是肯定的，否则答案为否定。

`javax.swing.plaf.UIResource` 接口是一个把对象标记为 UI 资源的标记接口。`javax.swing.plaf` 包提供 `ColorUIResource` 和 `FontUIResource` 扩展资源类并实现 `UIResource` 接口。例如，`ColorUIResource` 扩展 `Color` 类并实现 `UIResource` 接口。因此，`ColorUIResource` 的实例是用 `UIResource` 接口标记的颜色。

我们再回到最初的问题，如果这些资源是 `UIResource` 的实例的话，`Swing` 将只替换 UI 资源。

图 7-7 中示出的小应用程序包含了一个按钮，其前景色可以作为 `Color` 的一个实例来设置

① 通常通过改变界面样式来改变 UI 代表。

(如果其中的 UIResource 复选框未选取的话), 也可以作为一个 UI 资源来设置 (如果选取了其中的 UIResource 复选框的话)。这个小应用程序还为选取 Motif 或 Java 的界面样式提供了单选按钮。

图 7-7 左上下图给出了当按钮的前景色作为一种颜色设置时改变界面样式的效果; 对右上下图来说, 按钮的颜色是作为 ColorUIResource 的一个实例来指定的。

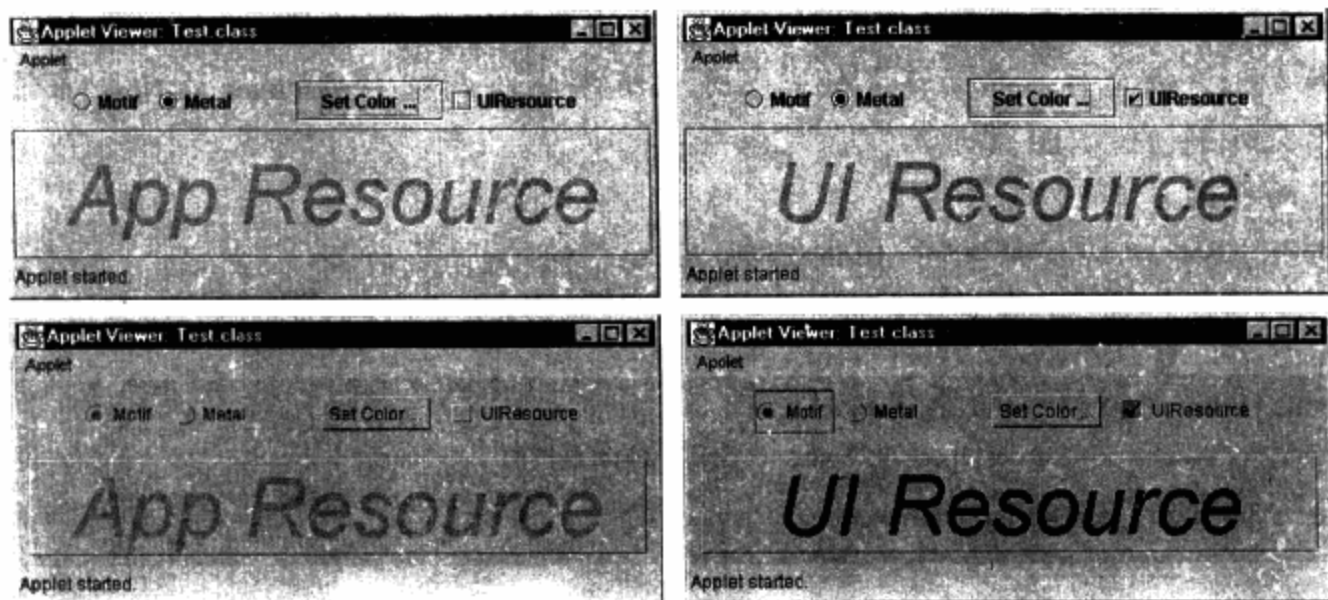


图 7-7 UI 资源

这个小应用程序在 Set Color ... 按钮和 UIResource 复选框上都添加了动作监听器。这两个监听器都调用了这个小应用程序的 updateButtonColor 方法, 该方法根据从一个颜色选取器中选取的颜色设置这个按钮的颜色。根据 UIResource 复选框是否选取, 这个颜色以一个 UI 资源或一种颜色的形式指定。

```
public class Test extends JApplet {
    private JButton button = new JButton ("App Resource");
    ...
    class ControlPanel extends JPanel {
        boolean resource = false;
        JButton colorSetButton = new JButton ("Set Color ...");
        JCheckBox checkBox = new JCheckBox ("UIResource");
        ...
        public ControlPanel () {
            ...
            colorSetButton.addActionListener (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    updateButtonColor ();
                }
            });
            checkBox.addActionListener (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    resource = checkBox.isSelected ();
                    updateButtonColor ();
                }
            });
        }
        private void updateButtonColor () {
            Color c = JColorChooser.showDialog (
                getContentPane (), // parent component
                "Choose a Color", // title
```

```

        setBackground ()); // initial color
    if (resource) {
        button.setText ("UI Resource");
        button.setForeground (new ColorUIResource (c));
    }
    else {
        button.setText ("App Resource");
        button.setForeground (c);
    }
}
...
}

```

例 7-4 完整地列出了图 7-7 示出的小应用程序的代码。

例 7-4 UI 资源

```

import javax.swing. * ;
import java.awt. * ;
import java.awt.event. * ;
import java.util. * ;

import java.swing.plaf.motif.MotifLookAndFeel;
import javax.swing.plaf.metal.MetalLookAndFeel;

import javax.swing.plaf.ColorUIResource;

public class Test extends JApplet {
    private JButton button = new JButton ("App Resource");

    public void init () {
        Container contentPane = getContentPane ();

        contentPane.add (new ControlPanel (), BorderLayout.NORTH);
        contentPane.add (button, BorderLayout.CENTER);
    }

    class ControlPanel extends JPanel {
        boolean resource = false;
        JButton colorSetButton = new JButton ("Set Color ...");
        JCheckBox checkBox = new JCheckBox ("UIResource");
        JRadioButton motifButton = new JRadioButton ("Motif"),
            metalButton = new JRadioButton ("Metal");

        public ControlPanel () {
            ActionListener listener = new RadioHandler ();
            ButtonGroup group = new ButtonGroup ();

            group.add (motifButton);
            group.add (metalButton);

            motifButton.addActionListener (listener);
            metalButton.addActionListener (listener);

            metalButton.setSelected (true);

            add (motifButton);
            add (metalButton);
            add (Box.createHorizontalStrut (25));
            add (colorSetButton);
        }
    }
}

```



```

add (checkBox);

Font buttonFont = button.getFont ();
button.setFont (new Font (buttonFont.getFamily (),
                          Font.ITALIC, 56));

colorSetButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        updateButtonColor ();
    }
});

checkBox.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        resource = checkBox.isSelected ();
        updateButtonColor ();
    }
});

private void updateButtonColor () {
    Color c = JColorChooser.showDialog (
        getContentPane (), // parent component
        "Choose a Color", // title
        getBackground ()); // initial color

    if (resource) {
        button.setText ("UI Resource");
        button.setForeground (new ColorUIResource (c));
    }
    else {
        button.setText ("App Resource");
        button.setForeground (c);
    }
}

class RadioHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JRadioButton src = (JRadioButton) e.getSource ();

        try {
            if (src == motifButton)
                UIManager.setLookAndFeel (
                    "javax.swing.plaf." +
                    "motif.MotifLookAndFeel");
            else if (src == metalButton)
                UIManager.setLookAndFeel (
                    "javax.swing.plaf.metal." +
                    "MetalLookAndFeel");
        }
        catch (Exception ex) {
            ex.printStackTrace ();
        }

        SwingUtilities.updateComponentTreeUI (
            getContentPane ());
    }
}

```

7.2 Java 界面样式

Java 界面样式是用代码名 (Java) 命名的一种 Metal 界面样式, 是 Swing 小应用程序和应用程序的缺省跨平台界面样式。Java 界面样式提供了两个在 Swing 的其他标准界面样式中没有的属性: 客户属性和主题 (themes)。

对某些组件来说, 可以设置客户属性未影响这些组件在 Java 界面样式中的界面样式。使用主题可以通过定义在界面样式中通用的颜色和字体来定制 Java 界面样式的外观。

7.2.1 客户属性

客户属性是一些可以在运行时刻添加到一个轻量 Swing 组件上的“键/值”对; 有关客户属性的更多讨论参见 4.9 节“客户属性”。

Java 界面样式对某些可用于很多组件的客户属性是敏感的。表 7-1 中列出了这些属性和它们的数据类型, 以及它们的缺省值。

表 7-1 Java 界面样式的客户属性

客户属性	数据类型	缺省值
<code>JInternalFrame.isPalette</code>	boolean	FALSE
<code>JScrollBar.isFreeStanding</code>	boolean	TRUE
<code>JSlider.isFilled</code>	boolean	FALSE
<code>JToolBar.isRollover</code>	boolean	FALSE
<code>JTree.lineStyle</code>	String	"Horizontal"

`JInternalFrame.isPalette`——确定用于内部窗体的边框的类型。调色板边框 (当 `JInternalFrame.isPalette` 为 true 时使用) 比正规的内部窗体边框细[⊖]。

`JScrollBar.isFreeStanding`——如果一个滚动条是独立的, 在它的周围就有一个蚀刻的边框。如果一个滚动条不是独立的, 其边框则只在该滚动条的上边和左边绘制。从图 7-8 中可以看到 `freeStanding` 属性的效果。

`JSlider.isFilled`——填充的水平滑杆用一种颜色填充滑柄左边的滑槽。垂直滑杆用颜色填充滑柄下方的滑槽。

`JToolBar.isRollover`——影响工具栏按钮边框的绘制方式。如果这个属性为 true, 则当光标进入一个按钮时绘制按钮边框, 并当光标离开时擦除按钮边框。其效果与 Netscape Navigator 中的工具栏按钮类似。

`JTree.lineStyle`——影响一个树中节点之间的连线的绘制方式。

图 7-8 中示出的小应用程序允许其组件设置客户属性。

在图 7-8 的上图中, 组件的 boolean 客户属性全设置为 false, 而树的 `lineStyle` 属性设置为 None。图 7-8 下面的图中, boolean 客户属性设置为 true, 树的 `lineStyle` 设置为 Angled。

⊖ 注意, 由于一个程序错误, 这个调色板属性对 Swing 1.1 FCS 没有作用。

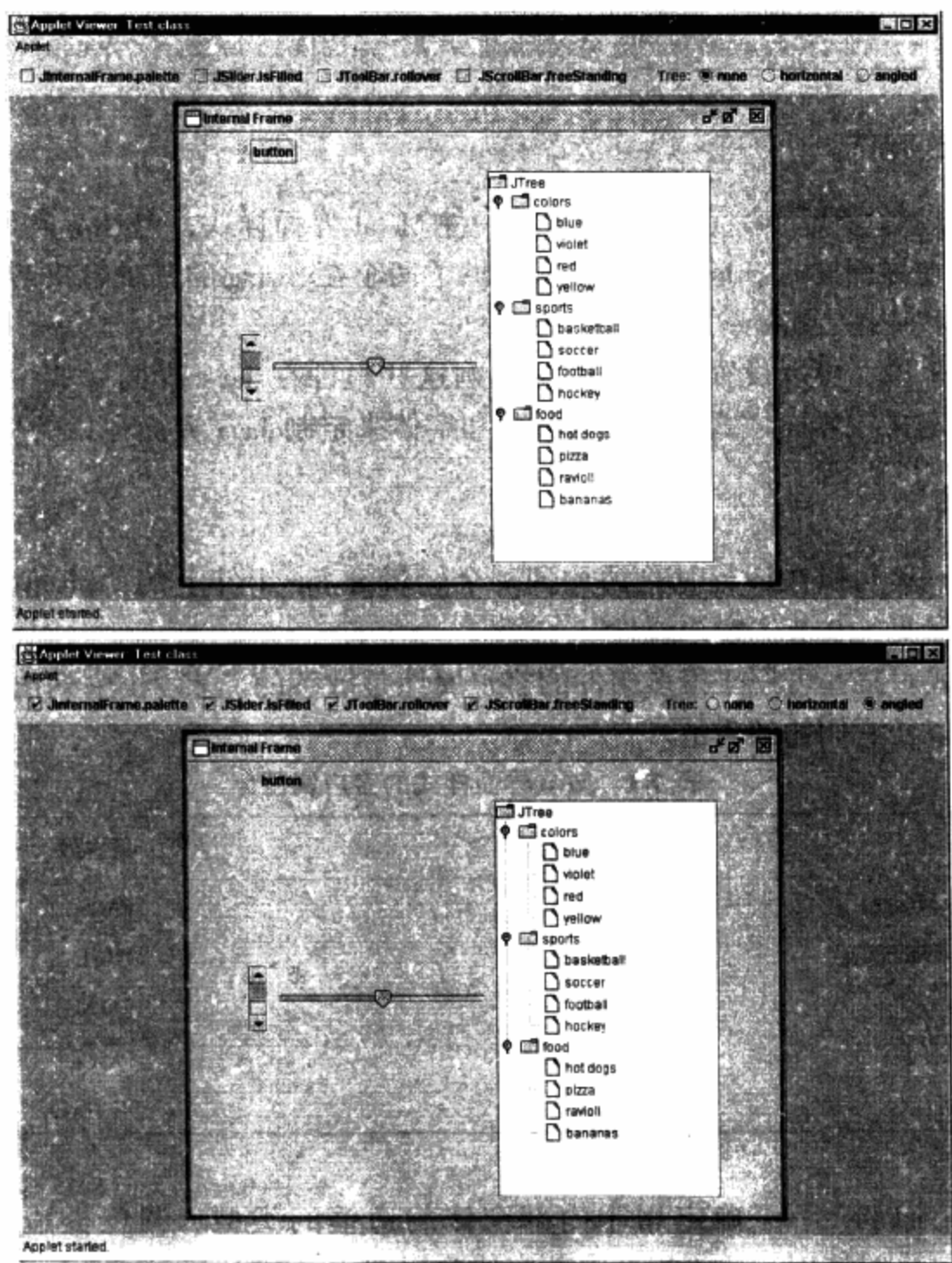


图 7-8 Metal 客户属性

这个小应用程序为其所包含的单选按钮和复选框添加了动作监听器。这些监听器根据已选取的复选框或单选钮来设置客户属性。注意，在指定一种客户属性后，必须调用 `repaint()` 或 `revalidate()`，因为在设置一种客户属性后，组件并不自动更新它们的外观。

```
class RadioButtonListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JRadioButton rb = (JRadioButton) e.getSource ();
        if (rb == none) {
            tree.putClientProperty (
                "JTree.lineStyle", "None");
        }
        if (rb == horizontal) {
            tree.putClientProperty (
                "JTree.lineStyle", "Horizontal");
        }
        if (rb == angled) {
            tree.putClientProperty (
                "JTree.lineStyle", "Angled");
        }
    }
}
```

```

        }
        tree.repaint ();
    }
}

class CheckBoxListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JCheckBox cb = (JCheckBox) e.getSource ();

        if (cb == palette) {
            palette.putClientProperty (
                "JInternalFrame.isPalette",
                new Boolean (cb.isSelected ()));

            jif.revalidate ();
        }
        else if (cb == filled) {
            slider.putClientProperty (
                "JSlider.isFilled",
                new Boolean (cb.isSelected ()));

            slider.repaint ();
        }
        else if (cb == rollover) {
            toolbar.putClientProperty (
                "JToolBar.isRollover",
                new Boolean (cb.isSelected ()));

            toolbar.repaint ();
        }
        else if (cb == freeStanding) {
            scrollbar.putClientProperty (
                "JScrollBar.isFreeStanding",
                new Boolean (cb.isSelected ()));

            scrollbar.repaint ();
        }
    }
}
}

```

例 7-5 中完整地列出了图 7-8 示出的小应用程序的代码。

例 7-5 Metal 客户属性

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    JDesktopPane desktopPane = new JDesktopPane ();

    JInternalFrame jif = new JInternalFrame (
        "Internal Frame", // title
        true, // resizable
        true, // closable
        true, // maximizable
        true); // iconifiable

    JScrollBar scrollbar = new JScrollBar ();
    JSlider slider = new JSlider ();
}

```

```

JToolBar toolbar = new JToolBar ();
JTree tree = new JTree ();

public void init () {
    Container contentPane = getContentPane ();

    jif.setPreferredSize (new Dimension (550, 450));
    jif.getContentPane ().setLayout (new FlowLayout ());
    jif.getContentPane ().add (new ComponentPanel ());

    desktopPane.setLayout (new FlowLayout ());
    desktopPane.add (jif);

    contentPane.add (new ControlPanel (), BorderLayout.NORTH);
    contentPane.add (desktopPane, BorderLayout.CENTER);
}

class ComponentPanel extends JPanel {
    public ComponentPanel () {
        JPanel panel = new JPanel ();

        setLayout (new BorderLayout ());
        add (toolbar, BorderLayout.NORTH);
        add (panel, BorderLayout.CENTER);

        panel.add (scrollbar);
        panel.add (slider);
        panel.add (new JScrollPane (tree));

        tree.setPreferredSize (new Dimension (200, 100));
        toolbar.add (new JButton ("button"));
    }
}

class ControlPanel extends JPanel {
    JCheckBox rollover = new JCheckBox (
        "JToolBar.rollover");
    JCheckBox palette = new JCheckBox (
        "JInternalFrame.palette");
    JCheckBox filled = new JCheckBox (
        "JSlider.isFilled");
    JCheckBox freeStanding = new JCheckBox (
        "JScrollBar.freeStanding");

    JRadioButton none = new JRadioButton ("none");
    JRadioButton horizontal = new JRadioButton ("horizontal");
    JRadioButton angled = new JRadioButton ("angled");

    public ControlPanel () {
        ActionListener checkBoxListener =
            new CheckBoxListener ();
        ActionListener radioButtonListener =
            new RadioButtonListener ();

        palette.addActionListener (checkBoxListener);
        filled.addActionListener (checkBoxListener);
        rollover.addActionListener (checkBoxListener);
        freeStanding.addActionListener (checkBoxListener);

        none.addActionListener (radioButtonListener);
        horizontal.addActionListener (radioButtonListener);
        angled.addActionListener (radioButtonListener);
    }
}

```

```

    ButtonGroup group = new ButtonGroup ();
    group.add (none);
    group.add (horizontal);
    group.add (filled);

    none.setSelected (true);
    freeStanding.setSelected (true);

    add (palette);
    add (filled);
    add (rollover);
    add (freeStanding);
    add (Box.createHorizontalStrut (10));
    add (new JLabel ("Tree: "));
    add (none);
    add (horizontal);
    add (angled);
}

class RadioButtonListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JRadioButton rb = (JRadioButton) e.getSource ();

        if (rb == none) {
            tree.putClientProperty (
                "JTree.lineStyle", "None");
        }
        if (rb == horizontal) {
            tree.putClientProperty (
                "JTree.lineStyle", "Horizontal");
        }
        if (rb == angled) {
            tree.putClientProperty (
                "JTree.lineStyle", "Angled");
        }
        tree.repaint ();
    }
}

class CheckBoxListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JCheckBox cb = (JCheckBox) e.getSource ();

        if (cb == palette) {
            palette.putClientProperty (
                "JInternalFrame.isPalette",
                new Boolean (cb.isSelected ()));

            jif.revalidate ();
        }
        else if (cb == filled) {
            slider.putClientProperty (
                "JSlider.isFilled",
                new Boolean (cb.isSelected ()));

            slider.repaint ();
        }
        else if (cb == rollover) {
            toolbar.putClientProperty (
                "JToolBar.isRollover",

```



```

MetalLookAndFeel.setCurrentTheme (new ExperimentalTheme ());

// Metal Look and Feel must be (re) loaded for the
// new theme to take effect ...

try {
    UIManager.setLookAndFeel (
        "javax.swing.plaf.metal.MetalLookAndFeel");
}
catch (IllegalAccessException e1) {}
catch (UnsupportedLookAndFeelException e2) {}
catch (InstantiationException e3) {}
catch (ClassNotFoundException e4) {}
}

class ExperimentalTheme extends DefaultMetalTheme {
    public FontUIResource getControlTextFont () {
        return new FontUIResource ("SanSerif",
                                    Font.BOLD + Font.ITALIC, 24);
    }
}

```

ExperimentalTheme 类扩展 DefaultMetalTheme 类。DefaultMetalTheme 类是 MetalTheme 的一个扩展，定义了缺省的 Metal 主题。ExperimentalTheme 重载 getControlTextFont 方法来指定绘制控件的字体。

注意，这个主题是用 MetalLookAndFeel.setCurrentTheme 方法设置的。为了使这个主题产生效果，在该方法之后必须安装这个 Java 的界面样式。

7.3 附加 UI

除了替换整个界面样式外，有时也希望为当前的界面样式添加一些功能。例如，使组件在激活时具有发声功能，比如：当按钮激活时发出啪嗒声或当滑杆器拖动时发出呼呼声。如果为此需要实现和安装一个全新的界面样式，那将是一件很费劲的事。

Swing 提供了一个复用界面样式，它实际上是多个界面样式的一个组合，该组合允许在当前界面样式上附加功能。通过调用 UIManager.getUI () 得到的组件的 UI 代表实际上是一个源于 javax.swing.multi 包的复用 UI 代表，如 MultiButtonUI 或 MultiLabelUI。下面介绍复用 UI 代表的工作方式。

复用 UI 代表实际上是当前界面样式的 UI 代表与已定义的附加界面样式的 UI 代表的组合体。当请求复用 UI 代表的图形属性（例如它的最大尺寸）时，它根据当前界面样式的 UI 代表返回值。当要求复用 UI 代表执行某些非图形操作（如安装或卸载一个组件）时，这个操作将针对当前界面样式的 UI 代表以及针对附加界面样式的 UI 代表来完成。

图 7-10 中示出的小应用程序中示出了一个附加界面样式，当光标进入一个按钮时，它显示一个句子。

例 7-7 列出了图 7-10 中示出的小应用程序的代码。

例 7-7 安装一个附加界面样式

```

import javax.swing.*;
import java.awt.*;

```

```

public class Test extends JApplet {
    public void init () {
        UIManager.addAuxiliaryLookAndFeel (
            new ExampleAuxiliaryLookAndFeel ());

        Container contentPane = getContentPane ();
        JButton button = new JButton ("button");

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);
    }
}

```

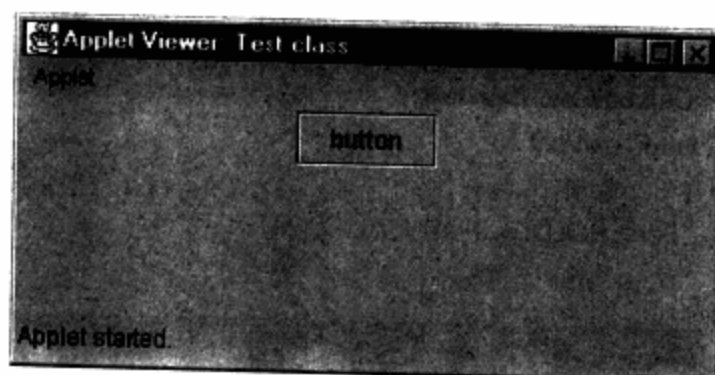


图 7-10 一个按钮的附加 UI

这个小应用程序用 `UIManager.addAuxiliaryLookAndFeel` 方法安装了 `ExampleAuxiliaryLookAndFeel` 的一个实例，作为一个附加界面样式。例 7-8 列出了 `ExampleAuxiliaryLookAndFeel` 类。

例 7-8 附加界面样式的例子

```

import java.awt. * ;
import javax.swing. * ;

public class ExampleAuxiliaryLookAndFeel extends LookAndFeel {
    public String getDescription () {
        return "example auxiliary look and feel";
    }

    public String getID () {
        return "example";
    }

    public String getName () {
        return "example auxiliary";
    }

    public boolean isNativeLookAndFeel () {
        return false;
    }

    public boolean isSupportedLookAndFeel () {
        return true;
    }

    public UIDefaults getDefaults () {
        UIDefaults table = new UIDefaults ();
        Object [] uiDefaults = {
            "ButtonUI", "AuxiliaryButtonUI"
        };
        table.putDefaults (uiDefaults);
    }
}

```

```
return table;
```

ExampleAuxiliaryLookAndFeel 类扩展 LookAndFeel 类，并实现由其超类定义的抽象方法。有关 LookAndFeel 类的更多信息参见 7.1.1 节“界面样式”。ExampleAuxiliaryLookAndFeel 在所有平台上都是支持的，在任何平台上都不是本地的。

这个 AuxiliaryButtonUI 类被 ExampleAuxiliaryLookAndFeel 类指定作为与按钮一起使用的 UI 代表。有关为界面样式指定缺省值的更多信息参见 7.1.2 节“界面样式缺省值”。

这个 AuxiliaryButtonUI 类实现 createUI 方法来返回 AuxiliaryButtonUI 的一个共享实例；对所有的按钮都使用同一个 AuxiliaryButtonUI 实例。

为了在按钮上添加或删除一个鼠标监听器，分别实现了 installUI 和 uninstallUI 方法。用一个空实现重载了 update 方法，因为其超类的 update 清除背景，并且这个附加按钮 UI 应当不影响其组件的显示外观。

例 7-9 一个附加 UI

```
import java.awt.*;
import java.awt.event.*;
import javax.accessibility.*;
import javax.swing.*;
import javax.swing.plaf.*;

public class AuxiliaryButtonUI extends ComponentUI {
    private static ComponentUicui = new AuxiliaryButtonUI ();
    private static AuxiliaryButtonMouseListener ml =
        new AuxiliaryButtonMouseListener ();
    // must be implemented
    public static ComponentUI createUI (JComponent c) {
        return cui;
    }
    public void installUI (JComponent c) {
        c.addMouseListener (ml);
    }
    public void uninstallUI (JComponent c) {
        c.removeMouseListener (ml);
    }
    public void update (Graphics g, JComponent c) {
        // don't want ComponentUI default behavior, which is
        // to clear the background
    }
}

class AuxiliaryButtonMouseListener extends MouseAdapter {
    public void mouseEntered (MouseEvent e) {
        JComponent c = (JComponent) e.getSource ();
        AccessibleContext ac = c.getAccessibleContext ();
        String role = ac.getAccessibleRole ().toString ();
        String name = ac.getAccessibleName ();
        System.out.println ("mouse entered component of type " +
            role + " named " + name);
    }
}
```

这个鼠标监听器通过获得按钮的访问信息并显示该信息来处理鼠标进入事件。

7.4 本章回顾

插入式界面样式使 Swing 应用程序看起来与本地应用程序一样。可插入性大概是 Swing 最显著的特征。插入式界面样式除了为一个小应用程序或应用程序设置界面样式外，大多数开发人员不会直接访问它。但是对插入式界面样式结构有一个基本理解是很重要的，因为插入式界面样式不仅可为一组组件设置界面样式，还能做其他更多的事情。

第二部分 Swing 组件

第 8 章 标签与按钮

Swing 的标签和按钮分别用 JLabel 和 JButton 类表示，它们是能够显示文本或图标的简单组件。

缺省时，标签没有边框，可以显示一个字符串、一个图标或同时显示字符串和图标。除了用于修饰文本域等不重要的小事情外，Swing 的标签还能起到图像画布（显示一个图像的组件）的作用。由于 AWT 的图像不是组件，不能把它们添加到一个容器中。因此，使用 AWT 的开发人员实现了各种不同的图像画布类；然而，在 Swing 中，可以把 JLabel 类当作图像画布使用^①。

按钮大概是使用最为普遍的用户界面组件。按钮通常带有某种边框，且可以被鼠标或快捷键激活。Swing 按钮比 Swing 标签要复杂得多，不仅因为能够激活它们来完成某个功能，而且很多其他 Swing 组件都是 AbstractButton 类的扩展，而 AbstractButton 类是 Swing 按钮的基类。

8.1 JLabel 与 JButton

尽管 JLabel 与 JButton 的很多方法（如表 8-1 中所示）具有完全相同的特征，但 JLabel 与 JButton 不是通过继承关系相联系的。结果是，尽管可以用非常相同的方式来操纵按钮和标签，但由于按钮和标签没有一个共同的基类，因此，不能用完全相同的方式来操纵它们。例如，用一个标签显示的文本与图标之间的间隔是可设置的，但这却不适用于按钮。同样，虽然按钮和标签都可以有一个不可激活的图标，但只有按钮可以有一个“滚过式”（rollover）图标，这种图标当光标进入按钮时才显示。

表 8-1 JLabel 与 JButton 重叠的方法

方法特征
protected int checkHorizontalKey (int, String)
protected int checkVerticalKey (int, String)
public AccessibleContext getAccessibleContext ()
public Icon getDisabledIcon ()
public int getHorizontalAlignment ()
public int getHorizontalTextPosition ()
public Icon getIcon ()
public String getText ()
public LabelUI getUI ()
public String getUIClassID ()
public int getVerticalAlignment ()
public int getVerticalTextPosition ()
public void setDisabledIcon (Icon)
public void setFont (Font)
public void setHorizontalAlignment (int)

① 有关图像画布的更多信息参见 4.3.1 “Swing 组件中的定制绘制”一节。

(续)

方法特征

```
public void setHorizontalTextPosition (int)
public void setIcon (Icon)
public void setText (String)
public void setVerticalAlignment (int)
public void setVerticalTextPosition (int)
public void updateUI
```

由一个按钮或标签显示的文本和/或图标统称作这个组件的内容。JLabel 和 JButton 都提供了控制排放它们的内容的方法，如图 8-1 所示。

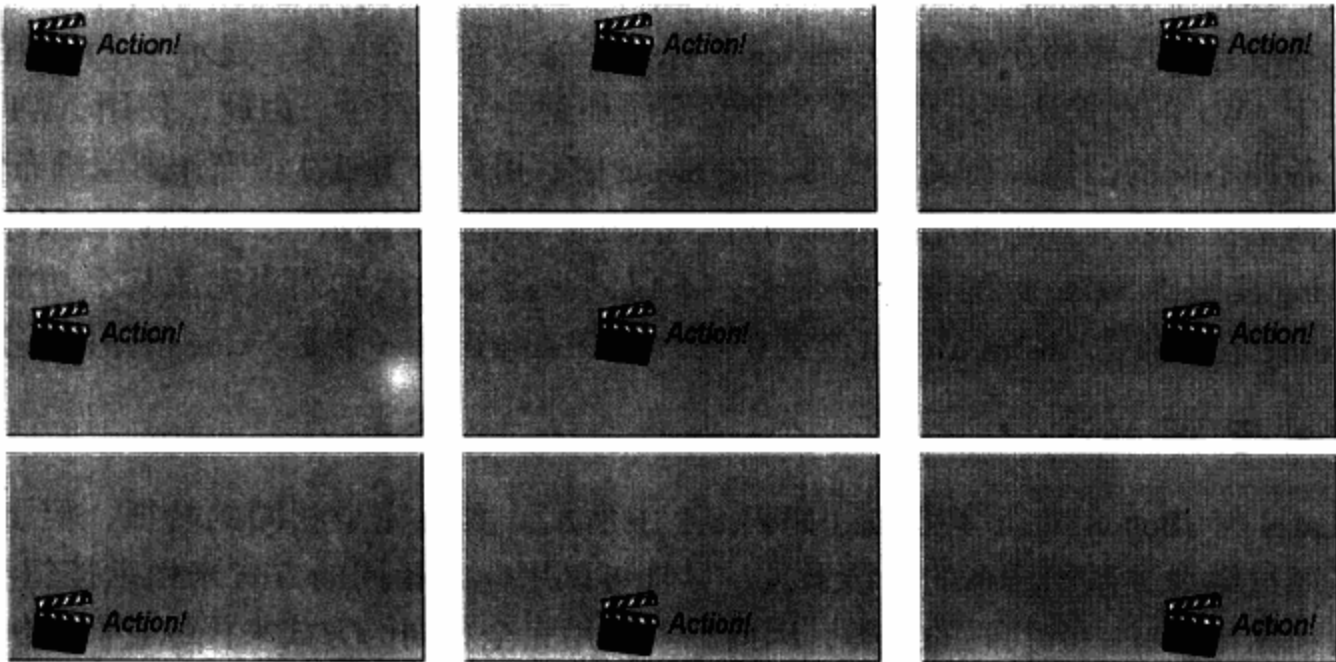


图 8-1 排列按钮和图标的内容

另外，对按钮和标签都可指定文本相对于图标的位置，如图 8-2 所示。Swing 的按钮和标签有 81 种排列内容和文本位置的组合方式。

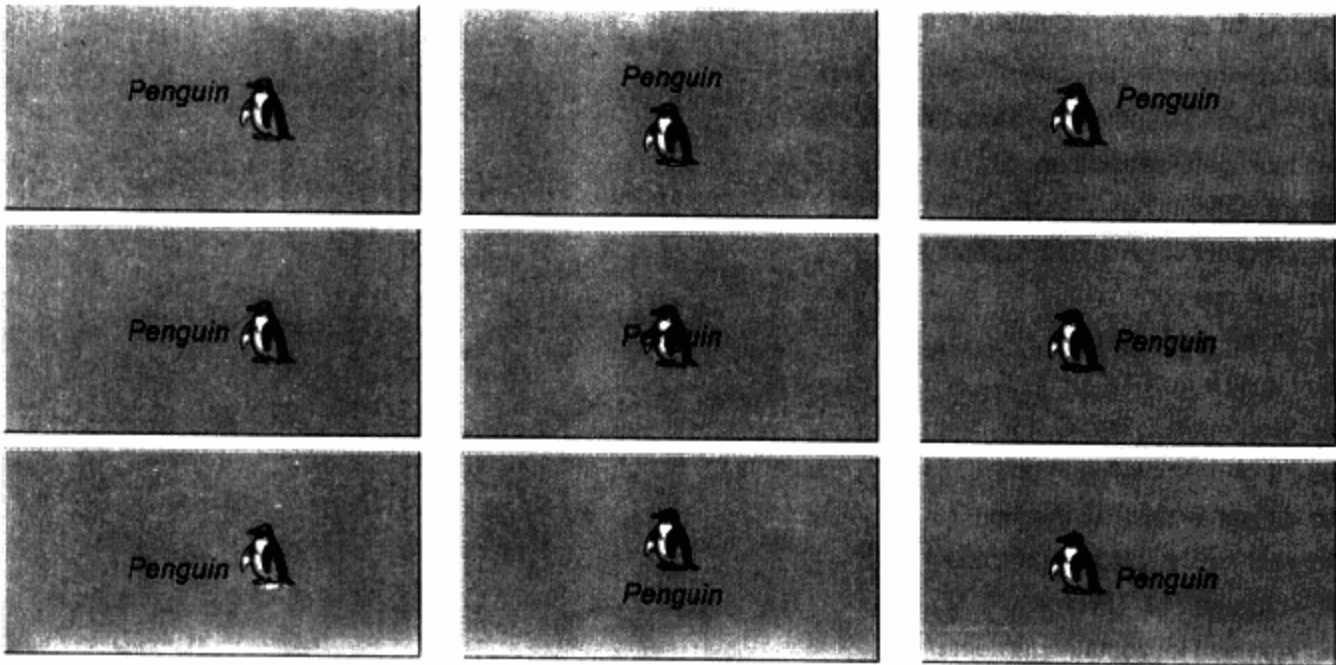


图 8-2 按钮和标签的文本位置

Swing 提示

Swing 的标签和按钮不是通过继承关系相联系的

虽然在显示文本和图标方面，Swing 的按钮和标签有很多共同的特性，但 JLabel 和 Abstract-Button 类没有扩展一个封装它们的共同特性的基类。因此，虽然可以用一种类似的方式操纵标签和按钮，但操纵它们的方式不是完全相同的。

8.2 JLabel

Swing 的标签具有多种用途。例如，除了起到显示一个图标的作用外，标签还通常用来绘制 Swing 表格的单元格。图 8-3 左图示出的小应用程序包含两个标签——其中一个标签含有一个图标和一个文本，而另一个标签包含在一个 Swing 滚动窗格中。图 8-3 右图的小应用程序包含一个 Swing 表格，该表格使用了定制的单元格绘制器。这些单元格绘制器是 JLabel 类的扩展。

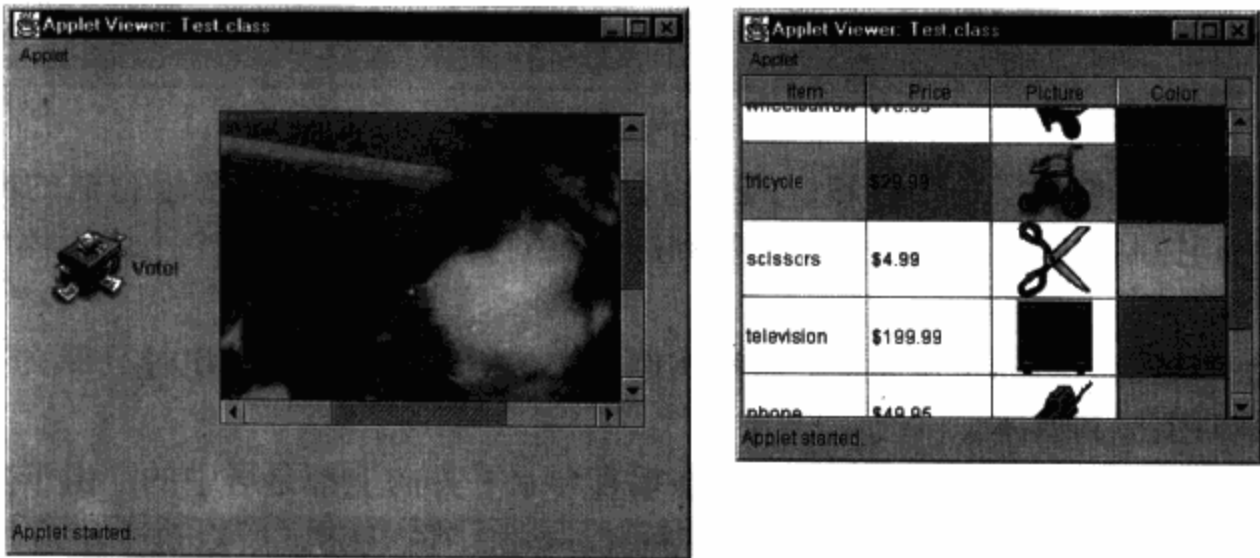


图 8-3 Swing 标签是多功能的

例 8-1 列出了图 8-3 左边的小应用程序的代码。

例 8-1 运行中的 JLabel

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane () ;
        JLabel imageOnly = new JLabel (new ImageIcon ("dogs.gif")) ;
        JLabel textAndImage = new JLabel ("Vote!",
            new ImageIcon ("ballot_box.gif"),
            JLabel.RIGHT) ;

        JScrollPane scrollPane = new JScrollPane (imageOnly) ;
        scrollPane.setPreferredSize (new Dimension (270, 200)) ;

        contentPane.setLayout (
            new FlowLayout (FlowLayout.CENTER, 25, 25)) ;

        contentPane.add (textAndImage) ;
        contentPane.add (scrollPane) ;
    }
}
```

这个小应用程序创建了 JLabel 的两个实例，并把其中之一指定为一个 JScrollPane 实例的视口（通过把这个标签的一个引用传送给滚动窗格的构造方法）。

这个小应用程序内容窗格的布局管理器设置为一个 FlowLayout 实例，另外，在这个小应用程序的内容窗格中还添加了一个 textAndImage 标签和一个滚动窗格。

8.2.1 内容排列

图 8-4 示出的小应用程序允许设置它所显示的标签的水平和垂直排列属性。

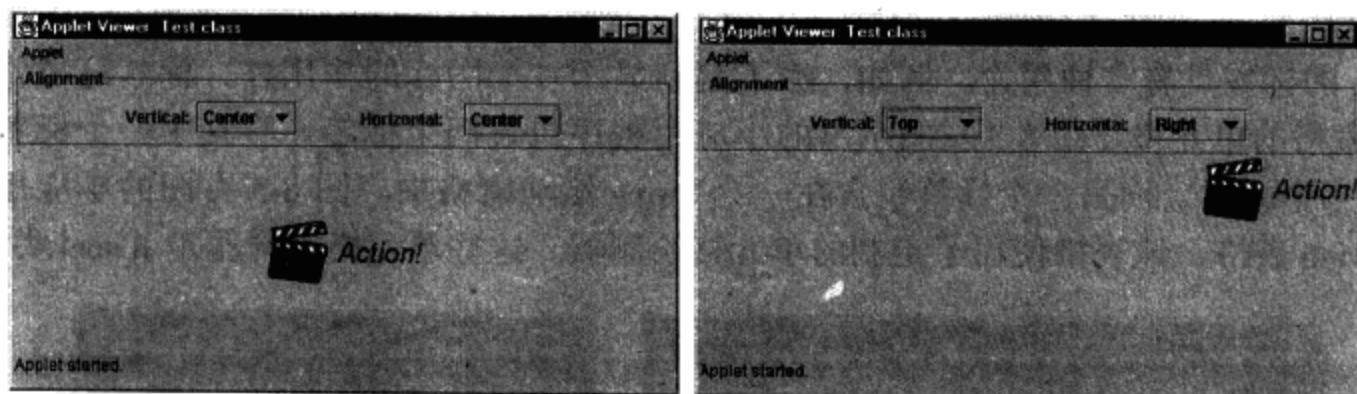


图 8-4 标签的排列方式

这个小应用程序创建了一个标签、两个组合框^①和一个包含这两个组合框的面板。这两个组合框添加到控制面板中，然后这个标签和这个控制面板都添加到这个小应用程序的内容窗格中。

由于标签中显示的图标是在标签之后被实例化的，所以这个标签创建时只带文本。然后再设置这个标签的图标和字体。

在这个小应用程序启动时，标签的内容在垂直和水平方向上都是居中的。由于水平和垂直排列的缺省值分别是 SwingConstants.LEFT 和 SwingConstants.CENTER，所以这个小应用程序把标签的水平排列值设置为 SwingConstants.CENTER。

```
public class Test extends JApplet implements SwingConstants {
    JLabel label = new JLabel ("Action!");
    JPanel controlPanel = new JPanel ();
    JComboBox alignmentHorizontal = new JComboBox ();
    JComboBox alignmentVertical = new JComboBox ();

    public void init () {
        Container contentPane = getContentPane ();
        ImageIcon icon = new ImageIcon ("slate.gif");

        label.setIcon (icon);
        label.setHorizontalAlignment (CENTER);
        label.setFont (new Font ("Times-Roman", Font.ITALIC, 20));

        setupComboBoxes ();
        setupControlPanel ();

        contentPane.setLayout (new BorderLayout ());
        contentPane.add (controlPanel, "North");
        contentPane.add (label, "Center");
        ...
    }
}
```

每个组合框都带有一个监听器，这个监听器为标签设置相应的排列方式。在每个组合框中

① 组合框在 18 章“组合框”中介绍。

选取的字符串都通过这个小应用程序的 `getSwingConstantByName` 方法转换成相应的 `SwingConstants` 常数。然后这个常数传送给 `JLabel.setVerticalAlignment()` 或 `JLabel.setHorizontalAlignment()`，以便设置这个标签的排列方式。

```
...
alignmentVertical.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        JComboBox b = (JComboBox) event.getSource ();
        String      s = (String) b.getSelectedItem ();
        int         c = getSwingConstantByName (s);

        label.setVerticalAlignment (c);
    }
});
alignmentHorizontal.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        JComboBox b = (JComboBox) event.getSource ();
        String      s = (String) b.getSelectedItem ();
        int         c = getSwingConstantByName (s);

        label.setHorizontalAlignment (c);
    }
});
} // end of init method
...
```

注意，在设置这个标签的排列属性后，不需要重画它，因为与 `JLabel` 的大多数属性一样，排列属性会导致重画标签[○]。

例 8-2 列出了图 8-4 中示出的小应用程序的代码。

例 8-2 设置 Swing 标签的排列属性

```
import java.net.URL;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet implements SwingConstants {
    JLabel label = new JLabel ("Action!");
    JPanel controlPanel = new JPanel ();
    JComboBox alignmentHorizontal = new JComboBox ();
    JComboBox alignmentVertical = new JComboBox ();

    public void init () {
        Container contentPane = getContentPane ();
        ImageIcon icon = new ImageIcon ("slate.gif");

        label.setIcon (icon);
        label.setHorizontalAlignment (CENTER);
        label.setFont (new Font ("Times-Roman", Font.ITALIC, 20));
        label.setMaximumSize (new Dimension (0, 150));

        setupComboBoxes ();
    }
}
```

○ 所有影响一个组件外观的属性的改变都将导致重画该组件。

```

        setupControlPanel ();

        contentPane.setLayout (new BorderLayout ());
        contentPane.add (controlPanel, "North");
        contentPane.add (label, "Center");

        alignmentVertical.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {
                JComboBox b = (JComboBox) event.getSource ();
                String      s = (String) b.getSelectedItem ();
                int          c = getSwingConstantByName (s);

                label.setVerticalAlignment (c);
            }
        });

        alignmentHorizontal.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {
                JComboBox b = (JComboBox) event.getSource ();
                String      s = (String) b.getSelectedItem ();
                int          c = getSwingConstantByName (s);

                label.setHorizontalAlignment (c);
            }
        });
    }

    void setupComboBoxes () {
        alignmentVertical.addItem ("Top");
        alignmentVertical.addItem ("Center");
        alignmentVertical.addItem ("Bottom");

        alignmentHorizontal.addItem ("Left");
        alignmentHorizontal.addItem ("Center");
        alignmentHorizontal.addItem ("Right");

        alignmentVertical.setSelectedItem (
            getSwingConstantName (
                label.getVerticalAlignment ()));
        alignmentHorizontal.setSelectedItem (
            getSwingConstantName (
                label.setHorizontalAlignment ()));
    }

    void setupControlPanel () {
        controlPanel.setBorder (
            BorderFactory.createTitledBorder ("Alignment"));

        controlPanel.add (new JLabel ("Vertical:"));
        controlPanel.add (alignmentVertical);
        controlPanel.add (Box.createHorizontalStrut (5));
        controlPanel.add (Box.createHorizontalStrut (25));

        controlPanel.add (new JLabel ("Horizontal:"));
        controlPanel.add (Box.createHorizontalStrut (5));
        controlPanel.add (alignmentHorizontal);
    }

    int getSwingConstantByName (String s) {
        if (s.equalsIgnoreCase ("left"))      return LEFT;
        else if (s.equalsIgnoreCase ("center")) return CENTER;
        else if (s.equalsIgnoreCase ("right")) return RIGHT;
    }

```

```

else if (s.equalsIgnoreCase ("top"))      return TOP;
else if (s.equalsIgnoreCase ("bottom"))   return BOTTOM;

return -1;
}

String getSwingConstantName (int c) {
    if (c == LEFT)      return "Left";
    else if (c == CENTER) return "Center";
    else if (c == RIGHT) return "Right";
    else if (c == TOP)   return "Top";
    else if (c == BOTTOM) return "Bottom";

    return "undefined";
}
}

```

8.2.2 文本的位置

图 8-5 中示出的小应用程序可以设置标签文本相对于图标的位置。

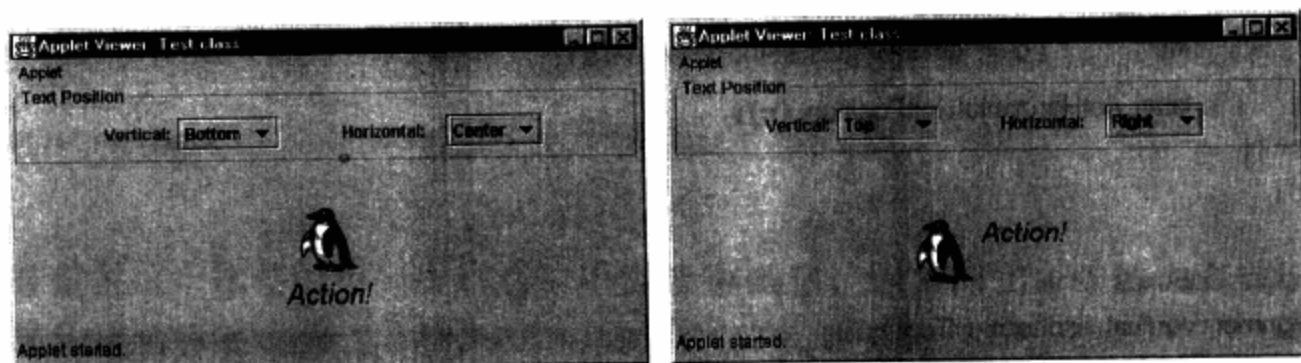


图 8-5 设置文本相对于图标的位置

由于图 8-5 中示出的小应用程序与图 8-4 中的小应用程序几乎完全相同，所以我们不再对例 8-3 中列出的这个小应用程序的代码进行讨论。图 8-5 中的小应用程序与图 8-4 中的小应用程序的不同之处在于它设置标签的水平 and 垂直文本位置属性，而不是设置排列方式。

例 8-3 设置标签的文本位置

```

import java.net.URL;
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.border. * ;

public class Test extends JApplet implements SwingConstants {
    JLabel label = new JLabel ("Action!");
    JPanel controlPanel = new JPanel ();
    JComboBox alignmentHorizontal = new JComboBox ();
    JComboBox alignmentVertical = new JComboBox ();

    public void init () {
        Container contentPane = getContentPane ();
        ImageIcon icon = new ImageIcon ("penguin.gif");

        label.setIcon (icon);
        label.setHorizontalTextPosition (CENTER);
        label.setFont (new Font ("Times-Roman", Font.ITALIC, 20));
    }
}

```

```

        setupComboBoxes ();
        setupControlPanel ();

        label.setHorizontalAlignment (JLabel.CENTER);
        label.setVerticalAlignment (JLabel.CENTER);

        contentPane.setLayout (new BorderLayout ());
        contentPane.add (controlPanel, "North");
        contentPane.add (label, "Center");

        alignmentVertical.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {
                JComboBox b = (JComboBox) event.getSource ();
                String      s = (String) b.getSelectedItem ();
                int         c = getSwingConstantByName (s);

                label.setVerticalTextPosition (c);
            }
        });

        alignmentHorizontal.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {
                JComboBox b = (JComboBox) event.getSource ();
                String      s = (String) b.getSelectedItem ();
                int         c = getSwingConstantByName (s);

                label.setHorizontalTextPosition (c);
            }
        });
    }

    void setupComboBoxes () {
        alignmentVertical.addItem ("Top");
        alignmentVertical.addItem ("Center");
        alignmentVertical.addItem ("Bottom");

        alignmentHorizontal.addItem ("Left");
        alignmentHorizontal.addItem ("Center");
        alignmentHorizontal.addItem ("Right");

        alignmentVertical.setSelectedItem (
            getSwingConstantName (
                label.getVerticalTextPosition ());
        );
        alignmentHorizontal.setSelectedItem (
            getSwingConstantName (
                label.getHorizontalTextPosition ());
        );
    }

    void setupControlPanel () {
        controlPanel.setBorder (
            BorderFactory.createTitledBorder ("Text Position"));

        controlPanel.add (new JLabel ("Vertical:"));
        controlPanel.add (alignmentVertical);
        controlPanel.add (Box.createHorizontalStrut (5));

        controlPanel.add (Box.createHorizontalStrut (25));
        controlPanel.add (new JLabel ("Horizontal:"));
        controlPanel.add (Box.createHorizontalStrut (5));
        controlPanel.add (alignmentHorizontal);
    }

    int getSwingConstantByName (String s) {

```

```

    if (s.equalsIgnoreCase ("left")) return LEFT;
    else if (s.equalsIgnoreCase ("center")) return CENTER;
    else if (s.equalsIgnoreCase ("right")) return RIGHT;
    else if (s.equalsIgnoreCase ("top")) return TOP;
    else if (s.equalsIgnoreCase ("bottom")) return BOTTOM;

    return -1;
}

String getSwingConstantName (int c) {
    if (c == LEFT) return "Left";
    else if (c == CENTER) return "Center";
    else if (c == RIGHT) return "Right";
    else if (c == TOP) return "Top";
    else if (c == BOTTOM) return "Bottom";

    return "undefined";
}

```

8.2.3 图标/文本间隙

图 8-6 示出的小应用程序可以设置标签中文本与图标之间的间隙。

这个小应用程序根据组合框中的值来设置标签文本与图标之间的间隙。这个组合框的子项监听器从组合框中提取被选取的字符串, 并使用 `Integer.parseInt()` 方法把该字符串转换为一个 `integer` 值。这个 `integer` 值然后传送给这个标签的 `setIconTextGap` 方法。

例 8-4 列出了图 8-6 所示的小应用程序的代码。

例 8-4 设置一个标签的图标/文本间隙

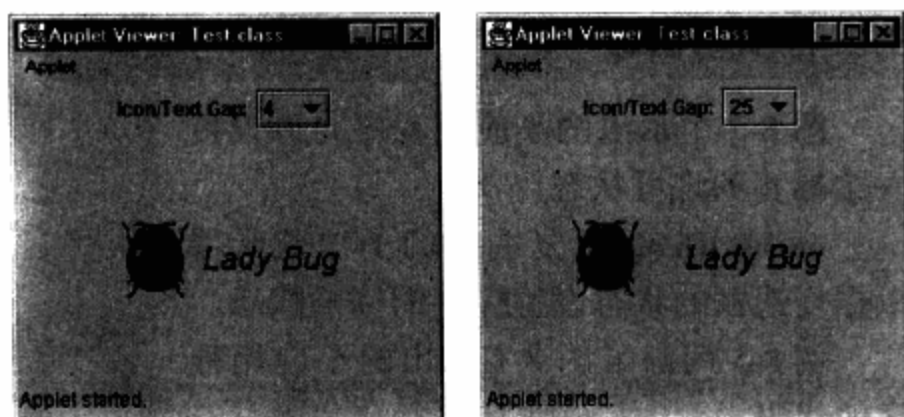


图 8-6 标签的图标/文本间隙

```

import java.net.URL;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet implements SwingConstants {
    public void init () {
        Container contentPane = getContentPane ();
        JComboBox iconTextGap = new JComboBox ();
        JPanel controlPanel = new JPanel ();
        ImageIcon icon = new ImageIcon ("ladybug.gif");

        final JLabel label = new JLabel ("Lady Bug", icon, CENTER);
        label.setFont (new Font ("Times-Roman", Font.ITALIC, 20));

        iconTextGap.addItem ("4");
        iconTextGap.addItem ("10");
        iconTextGap.addItem ("15");
        iconTextGap.addItem ("20");
        iconTextGap.addItem ("25");
    }
}

```



```

controlPanel.add (new JLabel ("Icon/Text Gap:"));
controlPanel.add (iconTextGap);

contentPane.setLayout (new BorderLayout ());
contentPane.add (controlPanel, "North");
contentPane.add (label, "Center");

iconTextGap.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        JComboBox b = (JComboBox) event.getSource ();
        String      s = (String) b.getSelectedItem ();
        int         gap = Integer.parseInt (s);

        label.setIconTextGap (gap);
    }
});

```

8.2.4 许可状态

图 8-7 中示出的小应用程序允许设置一个标签的许可状态。

enabledDisabled 复选框的子项监听器根据复选框的状态设置标签的许可状态。

例 8-5 列出了图 8-7 示出的小应用程序的代码。

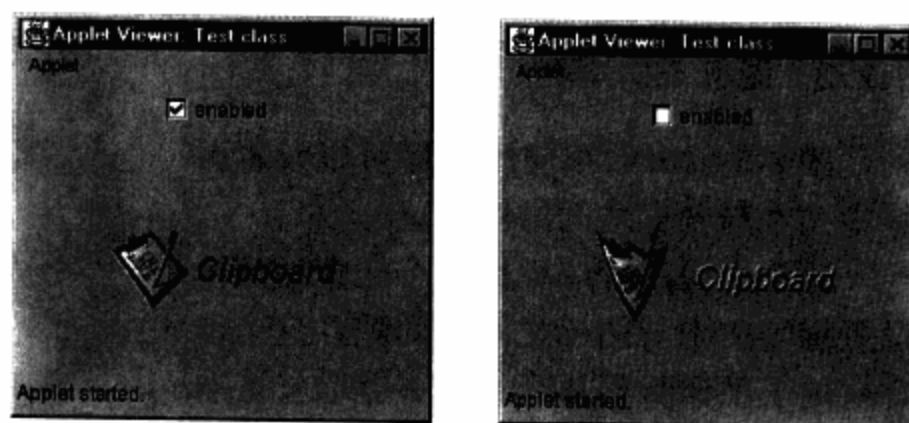


图 8-7 启用/禁用一个标签

例 8-5 设置一个标签的许可状态

```

import java.net.URL;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet implements SwingConstants {
    public void init () {
        final Container contentPane = getContentPane ();
        JCheckBox enabledDisabled = new JCheckBox ("enabled");
        JPanel controlPanel = new JPanel ();
        ImageIcon icon = new ImageIcon ("clipboard.gif");

        final JLabel label =
            new JLabel ("Clipboard", icon, CENTER);
        label.setFont (new Font ("Times-Roman", Font.ITALIC, 20));

        controlPanel.add (enabledDisabled);
        enabledDisabled.setSelected (true);

        contentPane.setLayout (new BorderLayout ());
        contentPane.add (controlPanel, "North");
        contentPane.add (label, "Center");

        enabledDisabled.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {

```

```
JCheckBox b = (JCheckBox) event.getSource ();
label.setEnabled (b.isSelected ());

    );
}
```

对那些支持禁用图标的 Swing 组件来说，缺省时，禁用图标是组件的一种图标形式，是通过 Swing 的 `GrayFilter` 类运行的。图 8-7 示出了通过一个 `GrayFilter` 实例运行一个图标的效果。

一些 Swing 组件支持定制的禁用图标。如果把例 8-5 中列出的小应用程序修改为设置标签的禁用图标，则使用的是禁用图标，而不是使用缺省的灰色图标，如图 8-8 所示。

组件总结 8-1 对 `JLabel` 类进行了总结。

组件总结 8-1 JLabel

模型：_____

UI 代表：swing.plaf.basic. LabelUI

绘制器：_____

编辑器：_____

激发的事件：PropertyChangeEvent

替代：java.awt. Label

类图：见图 8-9

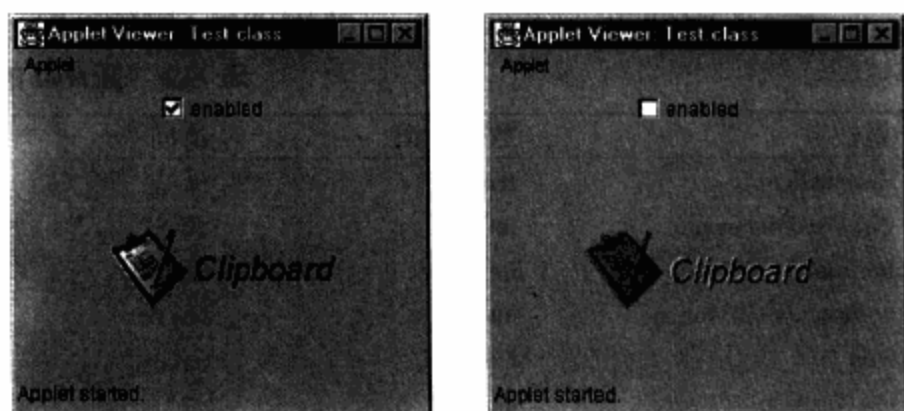


图 8-8 一个定制的禁用图标

图 8-9 示出了 `JLabel` 类的类图。`JLabel` 实现 `Accessible` 界面，以便通过一个 `AccessibleContext` 对象为可访问性工具提供信息。

与许多 Swing 组件一样，`JLabel` 也实现 `SwingConstants` 界面。由于实现了 `SwingConstants`，所以 `JLabel` 能够在不使用 `SwingConstants` 前缀的情况下指定 `SwingConstants` 中定义的常量。有关 `SwingConstants` 类及其使用方法的更多信息参见 6.4 节“`SwingConstants` 常量”。

`JLabel` 维护对它所显示的文本和图标的引用。`JLabel` 还维护 6 个用于跟踪某些属性的值，这些属性将在下一节“`JLabel` 属性”中讨论。

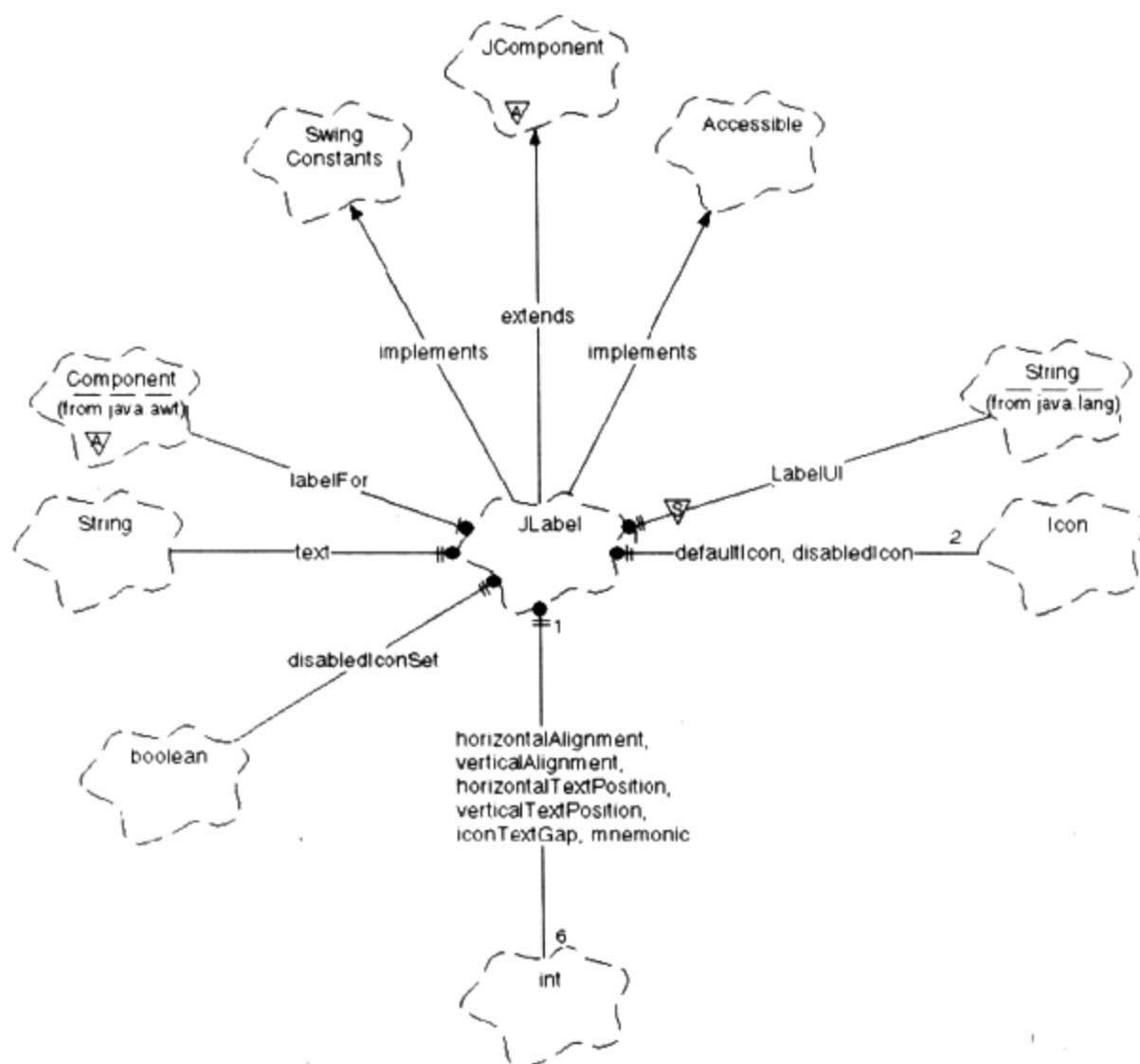


图 8-9 JLabel 类的类图

JLabel 还维护一个 Component 引用，这个引用用于标签的 labelFor 属性。有关 labelFor 属性的更多信息参见下一节“JLabel 属性”。标签还跟踪它们的 UI 代表的名称，及是否设置了禁用图标。

Swing 提示

Swing 标签具有多种用途

Swing 标签可以用于多种目的。除了只显示文本外，标签还可以用作图像画布；还可扩展 JLabel 类，以便为 JLabel 和 JTree 等 Swing 组件提供定制的单元绘制器。

8.2.5 JLabel 属性

表 8-2 中列出了 JLabel 实例维护的属性。

表 8-2 JLabel 的属性

属性名	数据类型	访问 ^①	类型 ^②	缺省值 ^③
disabledIcon	Icon	SG	B	灰色过滤图标
displayedMnemonic	int	SG	B	—
horizontalAlignment	int	CSG	B	LEFT
horizontalTextPosition	int	SG	B	RIGHT
icon	Icon	CSG	B	null
iconTextGap	int	SG	B	4 个像素
labelFor	Component	SG	B	null
text	String	CSG	B	""
verticalAlignment	int	SG	B	CENTER
verticalTextPosition	int	SG	B	CENTER

- ① C = 可在创建时设置/G = 获取方法/S = 设置方法
- ② B = 关联的（激发 PropertyChangeEvent）/Bool = boolean/C = 约束的/I = 索引的/S = 简单的
- ③ L&F = 与界面样式有关

disabledIcon——标签被禁用时显示的图标。缺省时，禁用图标是标签图标的洗白版本，这个版本是用 GrayFilter 获得的。可以为标签的禁用图标指定一个不同的图标。

displayedMnemonic——设置 labelFor 属性后显示的助记符。

horizontalAlignment——标签内容（即标签的文本和图标）的水平排列方式。可能的值是 JLabel.LEFT、JLabel.RIGHT 和 JLabel.CENTER。

horizontalTextPosition——文本相对于图标的水平位置；与水平排列方式的有效值相同。

icon——标签在启用时显示的图标。

iconTextGap——一个标签的文本与图标之间的间隙，以像素点为单位。

labelFor——一个组件，它在标签的助记符被键入时获得焦点。

text——一个标签显示的文本。

verticalAlignment——一个标签的内容的垂直排列方式。可能的值是 JLabel.TOP、JLabel.CENTER 和 JLabel.BOTTOM。

verticalTextPosition——文本相对于图标的垂直位置；与垂直排列方式的有效值相同。

JLabel 类维护的所有属性都有设置方法和获取方法，并且它们都是关联属性，即不论何时改变这些属性都激发属性变化事件。只有文本、图标的水平排列属性可以在创建时指定。

displayMnemonic 和 labelFor 属性是结合在一起使用的。如果设置了 displayMnemonic，则标签将使代表助记符的字符带下划线。如果还设置了一个 labelFor 组件，并且在标签所处窗口中的

任何组件具有焦点时键入了助记符, 则这个 `labelFor` 组件将得到焦点。当文本域等组件需要一个键盘助记符但又不能显示它时, 使用这个特性是比较方便的。

8.2.6 JLabel 事件

`JLabel` 主要是一个只显示的组件; `JLabel` 类只激发属性变化事件。当表 8-2 中列出的属性修改时, 则激发属性变化事件。

尽管 `JLabel` 类只激发属性变化事件, 但 `Swing` 标签能够激发多种事件, 从父组件事件到鼠标事件, 这是从 `JComponent` 类继承的特性。

缺省时, `JLabel` 不激发键击事件, 因此, 也就不接收键盘焦点。

8.2.7 JLabel 类总结

类总结 8-1 列出了 `JLabel` 的 `public` 和 `protected` 变量和方法。

类总结 8-1 JLabel

扩展: `JComponent`

实现: `SwingConstants`、`javax.accessibility.Accessible`

1. 构造方法

```
public JLabel ()
public JLabel (Icon)
public JLabel (Icon, int horizontalAlignment)
public JLabel (String)
public JLabel (String, Icon, int horizontalAlignment)
public JLabel (String, int horizontalAlignment)
```

`JLabel` 类提供了六个构造方法来产生各种配置的标签。构造方法所载入的 `integer` 值指定标签内容的水平排列方式。

注意, 其中一个既带一个字符串参数又带一个图标参数的构造方法还带了一个 `integer` 参数, 这个 `integer` 参数标识字符串和图标的排列方式。如果 `JLabel` (与 `JButton` 一样) 提供仅带一个字符串和一个图标的构造方法, 而水平排列采用某个缺省值, 那么情况就更好了。而目前的情况是, 如果一个 `JLabel` 实例是用一个字符串和一个图标构造的, 那么必须明确地设置水平排列方式。

图 8-10 示出了一个创建了四个 `JLabel` 实例的小应用程序。

这个小应用程序用一个无参数构造方法在左边创建了一个不可见的标签, 即这个标签既没有文本又没有图标, 因此在这个小应用程序中是不可见的。这个小应用程序还创建了一个只有文本的标签、一个只有图标的标签和一个既带文本又带图标的标签。

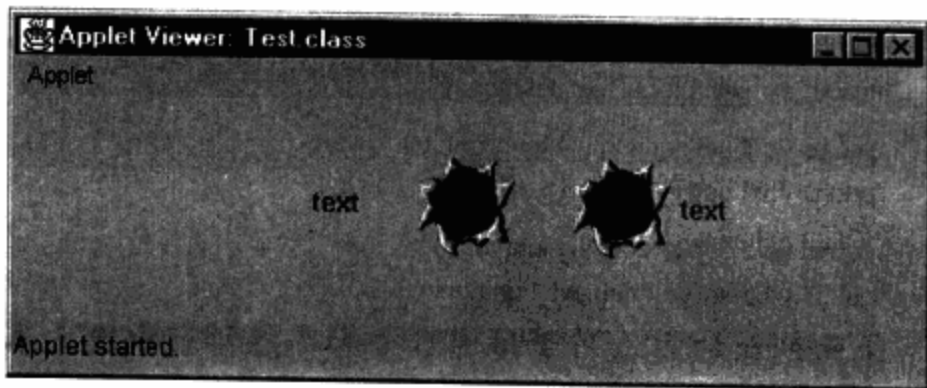


图 8-10 创建 JLabel 实例

例 8-6 中列出了图 8-10 示出的小应用程序的代码。

例 8-6 创建 JLabel 实例

```
import java.awt.*;
import java.awt.event.*;
```

```

import javax.swing.*;

public class Test extends JApplet {
    Icon icon = new ImageIcon ("icon.gif");

    JLabel defaultLabel = new JLabel (),
        textLabel = new JLabel ("text"),
        textIconLabel = new JLabel ("text", icon,
                                   SwingConstants.CENTER),
        iconLabel = new JLabel (icon);

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (
            new FlowLayout (FlowLayout.CENTER, 25, 25));

        contentPane.add (defaultLabel);
        contentPane.add (textLabel);
        contentPane.add (iconLabel);
        contentPane.add (textIconLabel);
    }
}

```

2. 方法

(1) 检验

```

protected int checkHorizontalKey (int, String)
protected int checkVerticalKey (int, String)

```

上面列出的两个方法确保传送给它们的 integer 值对指定标签内容的水平和垂直排列，以及对文本相对于图标的位置是有效值。表 8-2 中提供了一个有效值列表。如果传送给它们的参数值是无效的，则这两个方法都将弹出一个 `IllegalArgumentException` 异常信息，并且传送给这两个方法的字符串将用作这个异常的消息。如果这些值是有效的，则这两个方法正常返回。

这两个方法是 `protected` 类型的，这样它们能够被 `JLabel` 的扩展重载，然而，在实际应用中很少重载这两个方法。

(2) 排列方式和文本位置

```

public int getHorizontalAlignment ()
public int getVerticalAlignment ()
public int getHorizontalTextPosition ()
public int getVerticalTextPosition ()

public void setHorizontalAlignment ()
public void setHorizontalTextPosition ()

public void setVerticalAlignment ()
public void setVerticalTextPosition ()

```

上面列出的方法用于访问水平和垂直排列以及文本的位置。

(3) 属性访问方法

```

public Icon getDisabledIcon ()
public int getDisplayedMnemonic ()
public Icon getIcon ()
public int getIconTextGap ()
public Component getLabelFor ()
public String getText ()

public void setDisabledIcon (Icon)

```

```

public void setDisplayMnemonic (char)
public void setDisplayMnemonic (int)
public void setFont (Font)
public void setIcon (Icon)
public void setIconTextGap (int)
public void setLabelFor (Component)
public void setText (String)
protected String  paramString ()

```

上面列出的前两组方法是 JLabel 属性的访问方法。8.2.5 节“JLabel 属性”中列出并介绍了 JLabel 的属性。

JLabel 与其 AWT 对等类 java.awt.Label 在源代码方面是不兼容的。只有 getText() 和 setText() 两个方法在 java.awt.Label 和 JLabel 中是共有的。

java.awt.Label 有一个对标签所显示文本进行排列的排列属性。比较而言, JLabel 提供了两个属性, 它们有效地替代了 AWT 标签的排列属性。这两个属性是水平排列和垂直排列属性。因此, java.awt.Label 仅提供了一对方法来设置和获取标签的排列属性, 而 JLabel 为水平和垂直排列属性分别提供了设置方法和获取方法。

上面列出的 paramString 方法返回一个标签的字符串表示。

(4) 可访问性/插入式界面样式

```

public LabelUI getUI ()
public void setUI (LabelUI)
public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()

```

上面方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

8.3 按钮

按钮是 Swing 的重要组成部分; 一方面, 它们是简单的按压式按钮, 而另一方面, 它们又是 Swing 最重要的组成部分之一。

JButton 类实现了按钮抽象, 但它的功能几乎都是从 AbstractButton 类继承的。JButton 是八个扩展 AbstractButton 的 Swing 组件之一。这些 AbstractButton 的扩展是:

- JButton
- JToggleButton
- JCheckBox
- JRadioButton
- JMenuItem
- JMenu
- JRadioButtonMenuItem
- JCheckBoxMenuItem

另外, JButton 还是很多类的超类, UI 代表用这些类创建 UI 元素。例如, Motif 按钮和 Metal 组合框按钮都是 JButton 类的扩展。

图 8-11 中示出的小应用程序包含了上面列出的每一个类的至少一个实例。

注意 菜单和菜单项都是 AbstractButton 的扩展。AWT 菜单和菜单项不是 Compon-

ent^①类的扩展，因此不能以操纵其他类型的组件的方式操纵。例如，不能为 AWT 按钮或菜单项设置背景。另一方面，Swing 菜单和菜单项不仅是 Component 类的扩展，还是名副其实的按钮。这样，能对按钮做的任何操作也都能用于菜单或菜单项。

图 8-11 中示出的小应用程序中包含了一个 JScrollBar 实例^②，这不仅由于滚动条是按钮，还由于具有 Windows 界面风格的滚动条的两个箭头按钮都是 JButton 的扩展。

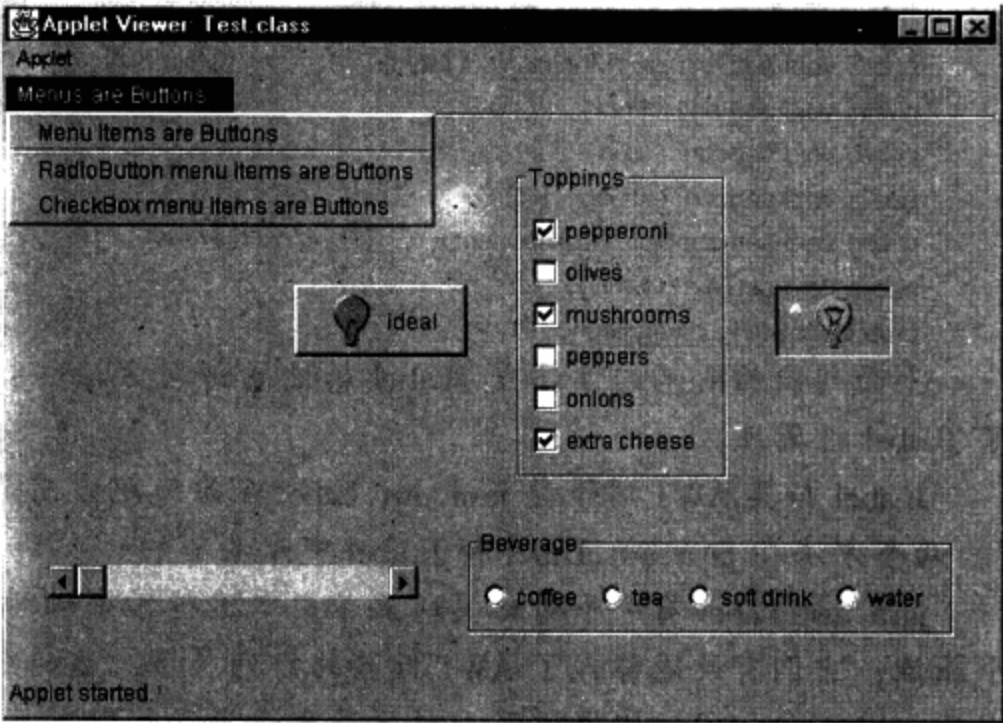


图 8-11 Swing 的按钮

按钮类的层次结构

图 8-12 中示出了 Swing 按钮的类图。

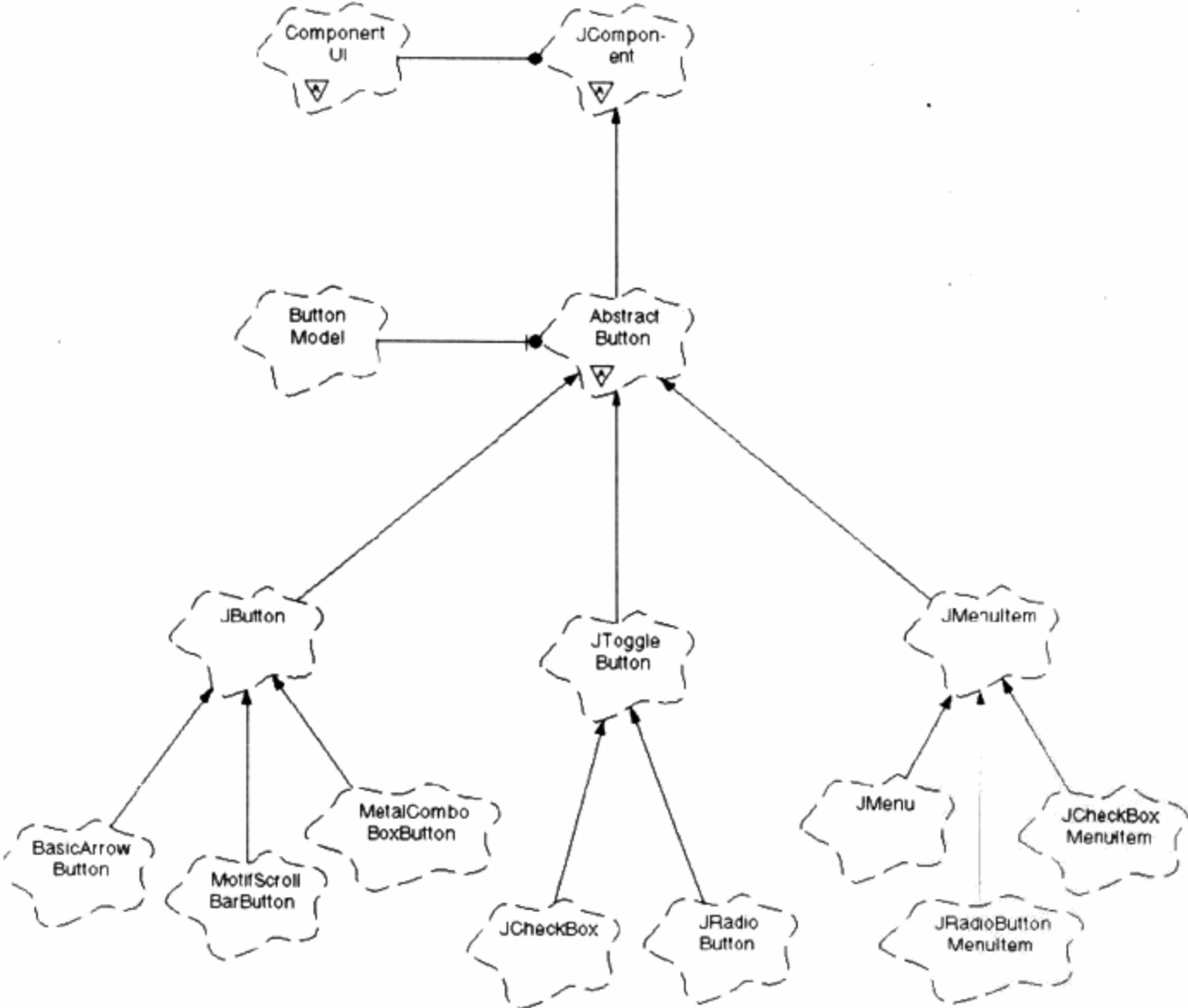


图 8-12 Swing 按钮类的层次结构

与所有其他的 Swing 轻量组件一样，按钮的 UI 代表是作为一个对 JComponent 类中的 Com-

① JButton —> AbstractButton —> JComponent —> Container —> Component
② 图 8-11 中的小应用程序未在书中列出，但在随带的盘中提供了。

ponentUI 的引用存储的。一个按钮的 UI 代表的实际类型取决于这个按钮的类型；例如，一个 JToggleButton 实例带有一个类型为 ToggleButtonUI 的 UI 代表。

一个按钮的模型是作为一个对 AbstractButton 类中的 ButtonModel 的引用存储的。与一个按钮的 UI 代表的情况一样，一个按钮的模型的类型取决于按钮的类型。

所有的 Swing 按钮都是 AbstractButton 类的扩展，这使得简单按钮、切换按钮、复选框、单选按钮，以及菜单和菜单项都具有按钮的共同特性。AbstractButton (JMenu 除外) 通常仅提供很多构造方法和少量的方法。

Swing 提示

AbstractButton 封装通用的按钮功能

在 AbstractButton 类中几乎封装了 Swing 按钮（包括菜单和菜单项）提供的所有功能。把通用功能封装到一个基类中是面向对象编程的原则之一。由于通用功能是在 AbstractButton 类中实现的，所以 AbstractButton 的扩展具有简单的实现，并且更为重要的是，它们都能以一种统一的方式操纵。

8.4 JButton

JButton 是一种按压式按钮，是 java.awt.Button 的替代品。与 Swing 标签一样，JButton 的实例能够显示一个文本或一个图标或同时显示两者。

与 AWT 按钮一样，当它们激活时，Swing 按钮激发动作事件。可以向一个按钮登记一个动作监听器。不论何时激活这个按钮，都调用这个监听器的 actionPerformed 方法。图 8-13 示出了监听一个 JButton 实例的动作事件的例子。

这个按钮的动作监听器在这个按钮的状态区显示这个按钮激活的次数。getActionCommand() 返回的字符串用于表示这个按钮；缺省时，getActionCommand() 返回这个按钮的文本。

例 8-7 中列出了图 8-13 示出的小应用程序。

例 8-7 一个按钮简单例子

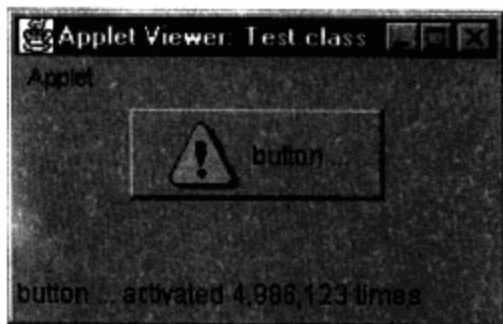


图 8-13 监听一个按钮的动作事件

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    JButton button = new JButton ("button ...",
                                new ImageIcon ("exclaim.gif"));
    int actCnt = 0;

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent event) {
                showStatus (event.getActionCommand () +
                           " activated " + actCnt + " times");
            }
        });
    }
}
```

```
actCnt + + ;
{
};
```

8.4.2 节“JButton 事件”中更详细地讨论了按钮事件。组件总结 8-2 中则进一步介绍了 JButton 类。

组件总结 8-2 JButton

模型: ButtonModel

UI 代表: swing.plaf.basic.ButtonUI

绘制器：_____

编辑器: _____

激发的事件: `ActionEvent/ChangeEvent/ItemEvent`

替代: `java.awt.Button`

类图： 图 8-14 示出了 JButton 类的类图。JButton 扩展 JComponent 类，它是一个轻量 Swing 组件。

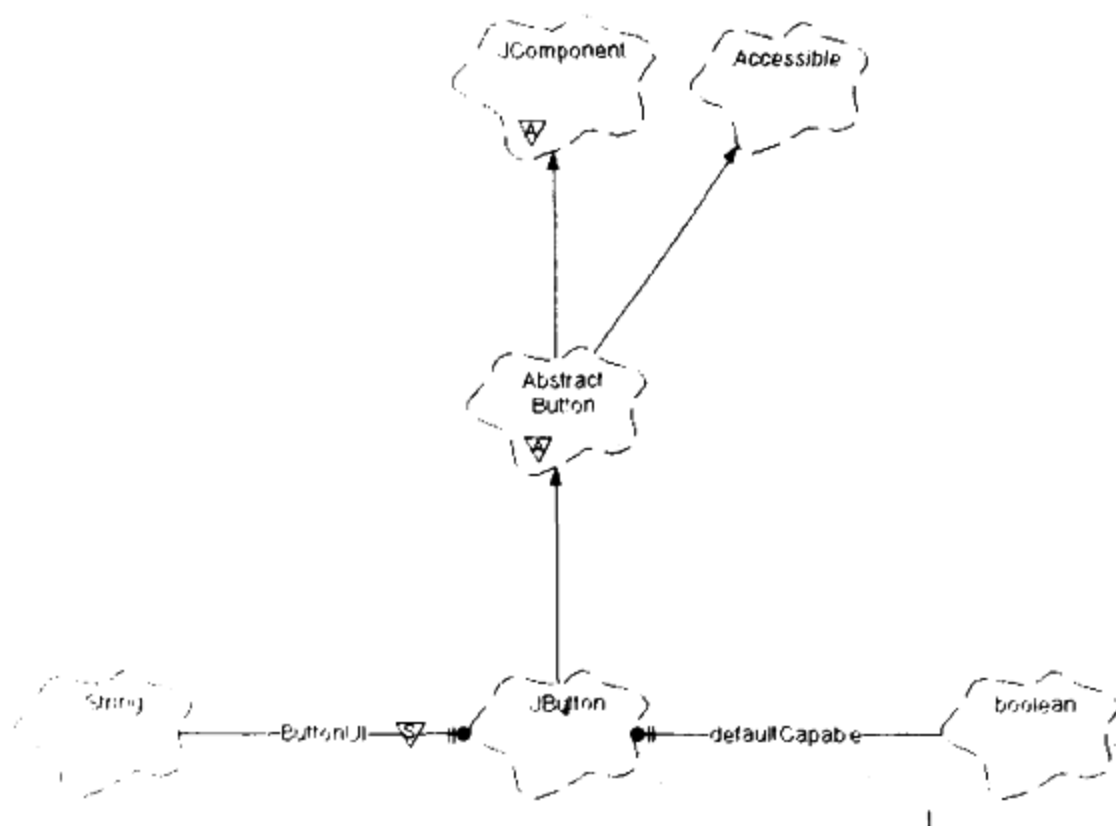


图 8-14 JButton 的类图

与所有 Swing 组件一样，JButton 实现 Accessible 接口，以便能够通过一个 AccessibleContext 对象为访问性工具提供信息。有关可访问性的更多信息，请参见 4.11 节“可访问性”。

JButton 几乎没有它自己的实现，它只有少量的构造方法及对可访问性的支持。JButton 实例跟踪它是否是一个根窗格^①中的缺省按钮，但是在大多数情况下，JButton 是一个非常薄的、放置在 AbstractButton 类上面的饰板。由于 AbstractButton 是所有动作发生的地方，因此，图 8-15 示出了 AbstractButton 类的类图。

① 参见例 8-10 “把一个按钮指定为缺省按钮”小节。

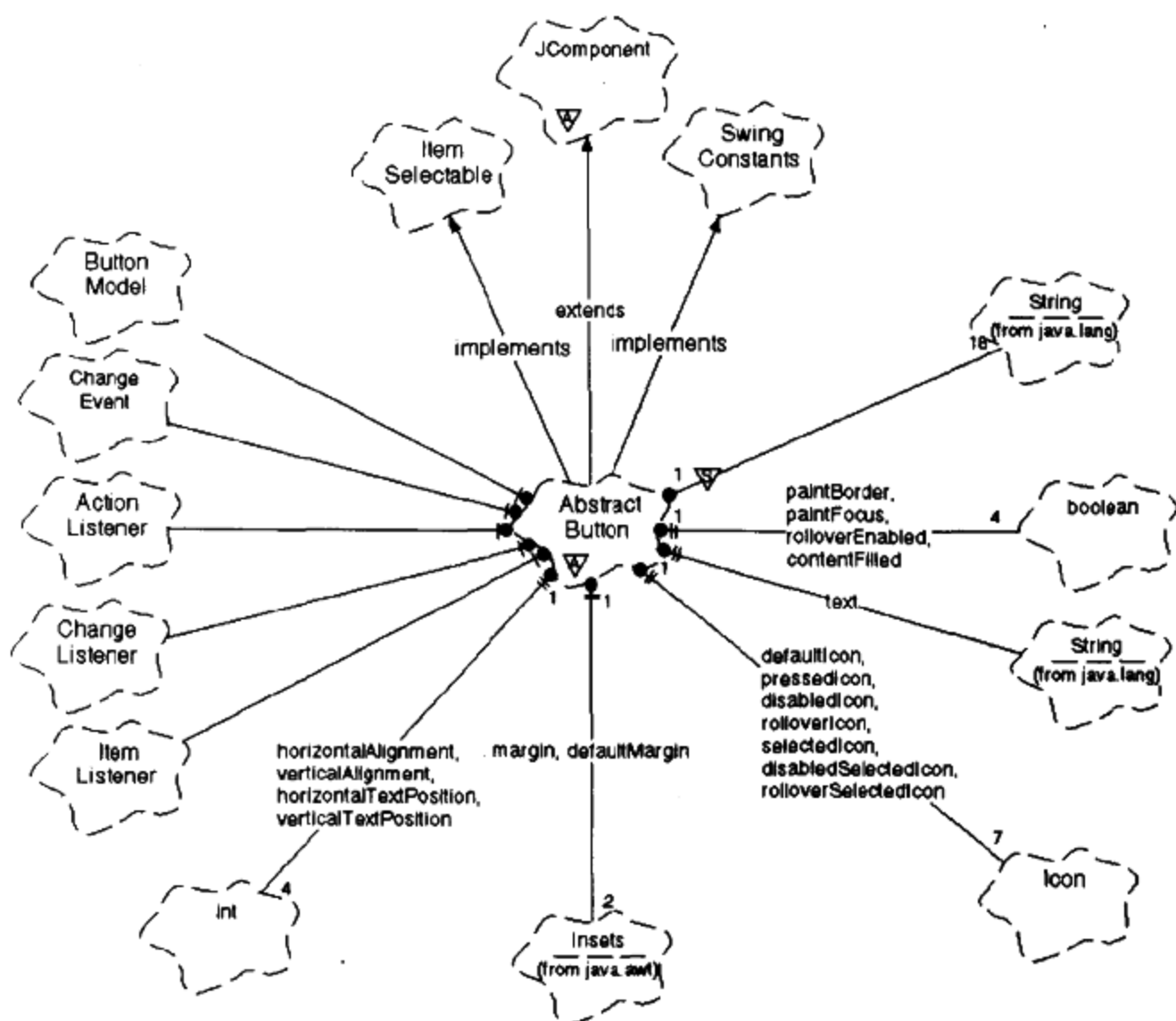


图 8-15 AbstractButton 类图

AbstractButton 类扩展 JComponent，并为其自身的便利^①实现 SwingConstants 接口。AbstractButton 还实现 java.awt.ItemSelectable，后者是一个为带有零个或多个可选子项的组件而设计的 AWT 接口。

AbstractButton 维护一个对其模型的 protected 引用。它的模型是一个实现 ButtonModel 接口的对象。AbstractButton 还维护对三个监听器的 protected 引用：ActionListener、ChangeListener 和 ItemListener。为了把模型事件传送给按钮自己的监听器，这三个监听器都添加到这个按钮的模型中。例如，在按钮的模型中添加一个按钮的子项监听器，且不论何时模型激发一个子项事件，模型都通知其监听器。则当接到通知时，监听器将向这个按钮的子项监听器传递事件。

除了事件源之外，变化事件没有联合关联状态。这样，每个 JButton 实例只需要一个 ChangeEvent 实例。这个按钮不论何时激发一个变化事件，这个实例都会重复使用。

AbstractButton 还维护了一些对 integer、boolean 和图标的私有引用，它们代表与 Swing 按钮相关联的属性。AbstractButton 的属性在下一节中讨论。

8.4.1 JButton 属性

表 8-3 中列出了 JButton 实例维护的属性。

① 即 AbstractButton 能够直接使用 SwingConstants 常数。

表 8-3 JButton 属性^①

属性名	数据类型	访问 ^②	类型 ^③	缺省值 ^④
actionCommand	String	SG	B	按钮文本
borderPainted	boolean	SG	B	true
defaultButton	boolean	G	B	false
defaultCapable	boolean	SG	B	true
disabledIcon	icon	SG	B	灰色图标
disabledSelectedIcon	icon	SG	B	灰色图标
focusPainted	boolean	SG	B	true
horizontalAlignment	int	SG	B	CENTER
horizontalTextPosition	int	SG	B	RIGHT
icon	icon	CSG	B	null
label	String	CSG	B	""
margin	Insets	SG	B	null
mnemonic	int	SG	B	0
model	ButtonModel	SG	B	DefaultButtonModel
pressedIcon	icon	SG	B	null
rolloverIcon	icon	SG	B	null
rolloverSelectedIcon	icon	SG	B	null
selected	boolean	SG	B	false
text	String	CSG	B	""
verticalAlignment	int	SG	B	CENTER
verticalTextPosition	int	SG	B	CENTER

① 用删除线划掉的属性表示被弃用的属性
② C = 可在创建时设置/G = 获取方法/S = 设置方法
③ B = 关联的（激发 PropertyChangeEvent）/Bool = boolean/C = 受约束的/I = 索引的/S = 简单的
④ I&F = 与界面样式有关

actionCommand—— 一个字符串，这个字符串标识与一个按钮相关的命令。缺省时，把一个按钮的动作命令设置为这个按钮的文本。

borderPainted—— 确定是否绘制一个按钮的边框。

defaultButton —— 指示一个按钮是否是一个根窗格的缺省按钮。

defaultCapable —— 指示一个按钮是否能作为一个根窗格的缺省按钮。

disabledIcon —— 当一个按钮禁用时显示的图标。缺省时，这个图标是该按钮的图标的灰色版本；但是，也可以显式地设置禁用按钮图标。

disabledSelectedIcon —— 这个属性未在 Swing 1.1 FCS 中使用。

focusPainted —— 确定当一个按钮具有焦点时，是否绘制一个焦点指示标志。

horizontalAlignment —— 一个按钮的内容（即它的文本和图标）的水平排列方式。可能的值是 JButton.LEFT、JButton.RIGHT 和 JButton.CENTER。

horizontalTextPosition —— 文本相对于图标的水平位置；有效值与水平排列方式的有效值相同。

icon —— 当启用一个标签时，这个标签显示的文本。

label —— 在一个按钮中显示的文本。这个 label 属性已不再使用；可用 text 属性代替之。

margin —— 一个按钮的边框与其内容之间的间隙。

mnemonic —— 用于激活一个按钮的键。

model —— 一个按钮的模型。缺省时，一个按钮的模型是一个 DefaultButtonModel 实例。

PressedIcon —— 当一个按钮按下时显示的一个图标。

rolloverIcon —— 当光标进入一个按钮时显示的一个图标。

rolloverSelectedIcon —— 这个属性目前未在 Swing FCS 中使用。

selected —— 这个属性是从 `AbstractAction` 继承而来的，对 `JButton` 的实例没有意义。可以选择 `AbstractButton` 的其他扩展；例如，当一个复选框打上了选中标志，它就是选取的。再比如，当一个菜单激活时，它就是已选取的。

text —— 在一个按钮中显示的文本。应该用 `text` 属性来替代过时的 `label` 属性。

verticalAlignment —— 文本相对于图标的垂直排列方式；其有效值与水平排列方式的有效值相同。

verticalTextPosition —— 文本相对于图标的垂直位置；其有效值与垂直排列方式的有效值相同。

表 8-3 中列出的大多数属性是从 `AbstractButton` 继承而来的。只有 `defaultButton` 属性（它实际上是在这个按钮的根窗格中维护的）和 `defaultCapable` 属性例外，它们是 `JButton` 自己定义的。

虽然 `JButton` 和 `JLabel` 不是通过继承关系联系起来的，但它们具有一些相同的属性：图标、文本、禁用图标、助记词、水平排列、水平文本位置、垂直排列和垂直文本位置。因此，`JButton` 和 `JLabel` 对上面列出的属性有一些（几乎）相同的访问方法。只有那些用了 `mnemonic` 属性的访问方法具有不同的用法。对标签来说，用 `getDisplayedMnemonic()` 来访问 `mnemonic` 属性，而对按钮则使用 `getMnemonic()`。

动作命令是一个字符串，它通常用于标识激发一个动作事件的按钮。如果一个按钮的动作命令没有显式地设置，其缺省值就是这个按钮的文本。动作命令的真正价值产生于这样一个事实：`java.awt.event.ActionEvent` 类带有一个 `getActionCommand` 方法，该方法返回与事件源相关的动作命令。因此，可以不采用下述方法：

```
//the hard way...
public void actionPerformed (ActionEvent e) {
    AbstractButton b = (AbstractButton) e.getSource ();
    if (b.getActionCommand () . equals ("cut")) {
        // do cut action
    }
    else {
        // do something else
    }
    ...
}
```

你可以不访问这个按钮，而直接访问这个动作命令：

```
//the easy way...
public void actionPerformed (ActionEvent e) {
    if (e.getActionCommand () . equals ("cut")) {
        // do cut action
    }
    else {
        // do something else
    }
    ...
}
```

`AbstractButton` 维护对不少于七个图标的引用，这足以保证表 8-4 中列出的图标的使用。

表 8-4 AbstractButton 维护的图标

图标	使用
图标	当按钮启用且未选取时显示
按下的图标	当按钮按下时显示
禁用图标	当按钮禁用时显示
已选取图标 ^①	当按钮选取时显示
滚过式图标	当光标进入这个按钮时显示
禁用的、已选取图标	未使用
滚过式、已选取图标	未使用

① 只与反复按钮和菜单项有关

后两种图标属性在 Swing 中没有任何使用。

Swing 提示

用动作命令来标识按钮

每个 AbstractButton 实例都维护一个称作动作命令的字符串标识符。缺省时，把一个按钮的动作命令设置为这个按钮中显示的文本，AbstractButton 还提供了一个方法来显式地设置动作命令字符串。动作命令字符串可被动作监听器用来标识哪个按钮激发了一个动作事件，因为 ActionEvent 类提供了一个返回与该事件相关的动作命令。这样，当一个按钮激发一个动作事件时，不必借助事件源来标识激发该动作的按钮。

8.4.2 JButton 事件

当 JButton 的实例激活时，它们激发动作事件；当它们的关联属性改变时，它们激发变化事件；而当它们的状态变化时，则激发变化事件。表 8-5 列出了按钮的状态。

表 8-5 按钮状态

状态	含义
待按下	光标处于按钮中时按下鼠标按钮。如果鼠标按钮在按钮中松开，这个按钮就被激活，否则它就恢复原态
启用	按钮能够改变状态
按下	按钮已经按下
进入	光标已经进入了按钮
已选取	按钮已经选取

3.2.4 节介绍了属性变化事件和变化事件。ActionListener 接口和 ActionEvent 类存在于 java.awt.event 包中。接口总结 8-1 对 ActionListener 接口进行了总结。

接口总结 8-1 ActionListener

public abstract void actionPerformed (ActionEvent)
actionPerformed 方法载入一个 ActionEvent 实例。类总结 8-2 对 ActionEvent 类进行了总结。

类总结 8-2 ActionEvent

扩展：java.awt.AWTEvent

1. 常量

```
public static final int ACTION_FIRST
public static final int ACTION_LAST
public static final int ACTION_PERFORMED

public static final int ALT_MASK
public static final int CTRL_MASK
public static final int META_MASK
public static final int SHIFT_MASK
```

上面列出的第一组常量代表动作事件的合法事件 ID。有关事件 ID 的更多信息，请参见《Java 2 图形设计，卷 I：AWT》。上面列出的第二组常量代表可能在动作事件属性发生时已经按下的修饰键。

2. 构造方法

```
public ActionEvent (Object source, int id, String command)
public ActionEvent (Object source, int id, String command, int modifiers)
```

动作事件是用事件源、事件 ID 和动作命令创建的。在构造方法中还指定了一个 integer 值，它代表在事件发生时被按住的修饰键。

3. 方法

```
public String getActionCommand ()
public int getModifiers ()
public String paramString ()
```

ActionEvent 类定义了获取动作命令字符串和修饰键的方法，还提供了一个 paramString 方法，用于创建由 toString 方法返回的字符串。

JButton 激发的所有事件都是从 AbstractButton 继承而来的。JButton 没有自己的事件。还应当注意的是，AbstractButton 具有激发子项事件的功能。能够被选取的 AbstractButton 扩展，如复选框和单选钮等激发子项事件。

图 8-16 中示出的小应用程序包含一个配备了一个变化监听器和一个动作监听器的按钮。当这个按钮激发变化或动作事件时，这些监听器显示与事件有关的信息。

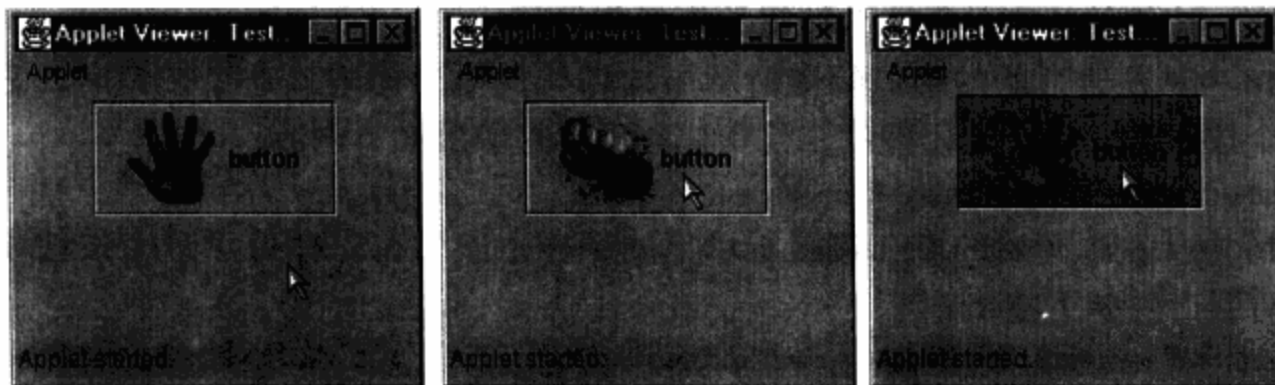


图 8-16 处理 JButton 事件

例 8-8 中列出了图 8-16 中所示的小应用程序的代码。

例 8-8 处理 JButton 事件

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {
    Icon icon = new ImageIcon ("icon.gif");
```



```

JButton button = new JButton ("button");

public Test () {
    Container contentPane = getContentPane ();

    button.setRolloverIcon (new ImageIcon ("punch.gif"));
    button.setIcon (new ImageIcon ("open_hand.gif"));

    contentPane.setLayout (new FlowLayout ());
    contentPane.add (button);

    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            System.out.println ("action!");
        }
    });
    button.addChangeListener (new ChangeListener () {
        public void stateChanged (ChangeEvent e) {
            System.out.println (getButtonState ());
        }
    });
}

private String getButtonState () {
    ButtonModel model = button.getModel ();
    String state = "Button State: ";

    state += model.isSelected () ? "selected" : "deselected";
    state += model.isPressed () ? ", pressed" :
        ", not pressed";
    state += model.isArmed () ? ", armed" : ", disarmed";
    state += model.isRollover () ? ", rollover" :
        ", not rollover";

    return state;
}

```

这个小应用程序的动作监听器简单地打印出字符串“action!”，而变化监听器则打印出这个按钮的状态。按钮的状态是从这个按钮的模型获得的。

如图 8-16 中从左边开始的第二幅图所示出的那样，当光标进入这个按钮时，显示光标进入图像，这个按钮激发一个变化事件，且变化监听器打印出如下内容：

```
Button State: not selected, not pressed, not armed, rollover
```

如图 8-16 中从左边开始的第三幅图所示出的那样，如果在光标处于这个按钮中时再按下鼠标，则监听器产生如下输出：

```
Button State: not selected, not pressed, armed, rollover
```

```
Button State: not selected, pressed, armed, rollover
```

如果在光标处于按钮中时再松开鼠标按钮，则监听器产生如下输出：

```
action!
```

```
Button State: not selected, not pressed, armed, rollover
```

```
Button State: not selected, not pressed, not armed, rollover
```

注意 在 Swing 1.1 中有一个错误，它使上面列出的状态输出中包含了“rollover”。除了上面的第一个输出外，这个按钮的状态都应该包含“not rollover”。

当光标离开这个按钮时，变化监听器输出如下内容：

```
Button State: not selected, not pressed, not armed, not rollover
```

8.4.3 JButton 类总结

类总结 8-3 中列出了 JButton 的 public 和 protected 变量，以及 JButton 的方法。

JButton 在具有大量特性的 AbstractButton 类基础上添加了大量构造方法。

类总结 8-3 JButton

扩展：AbstractButton

实现：javax.accessibility.Accessible

1. 构造方法

```
public JButton ()
public JButton (Icon)
public JButton (String)
public JButton (String, Icon)
```

图 8-17 中示出的小应用程序用上述四个构造方法创建了四个 JButton 实例。

与图 8-10 中的小应用程序用 JLabel 的无参数构造方法创建的标签不同，图 8-17 中用 JButton 的无参数构造方法创建的按钮是可见的，因为绘制了这个按钮的边框。

例 8-9 中列出了图 8-17 所示的小应用程序的代码。



图 8-17 创建 JButton 实例

例 8-9 创建 JButton 实例

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    Icon icon = new ImageIcon ("icon.gif");

    JButton noargButton = new JButton (),
        textButton = new JButton ("text"),
        textIconButton = new JButton ("text", icon),
        iconButton = new JButton (icon);

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (noargButton);
        contentPane.add (textButton);
        contentPane.add (iconButton);
        contentPane.add (textIconButton);
    }
}
```

2. 方法

(1) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
```

```
public void updateUI ()
protected String  paramString ()
```

上面列出的方法在 JComponent 的任意扩展中都能够找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。ParamString 方法返回一个按钮的字符串表示。

(2) 缺省按钮

```
public boolean isDefaultButton ()
public void setDefaultCapable ()
public boolean isDefaultCapable ()
```

缺省可用的按钮可以指定为一个根窗格的缺省按钮。缺省按钮通常以这样一种方式绘制，它们的外观有别于常规按钮并可用捷径键激活。缺省按钮的确切外观和行为取决于它的界面样式。

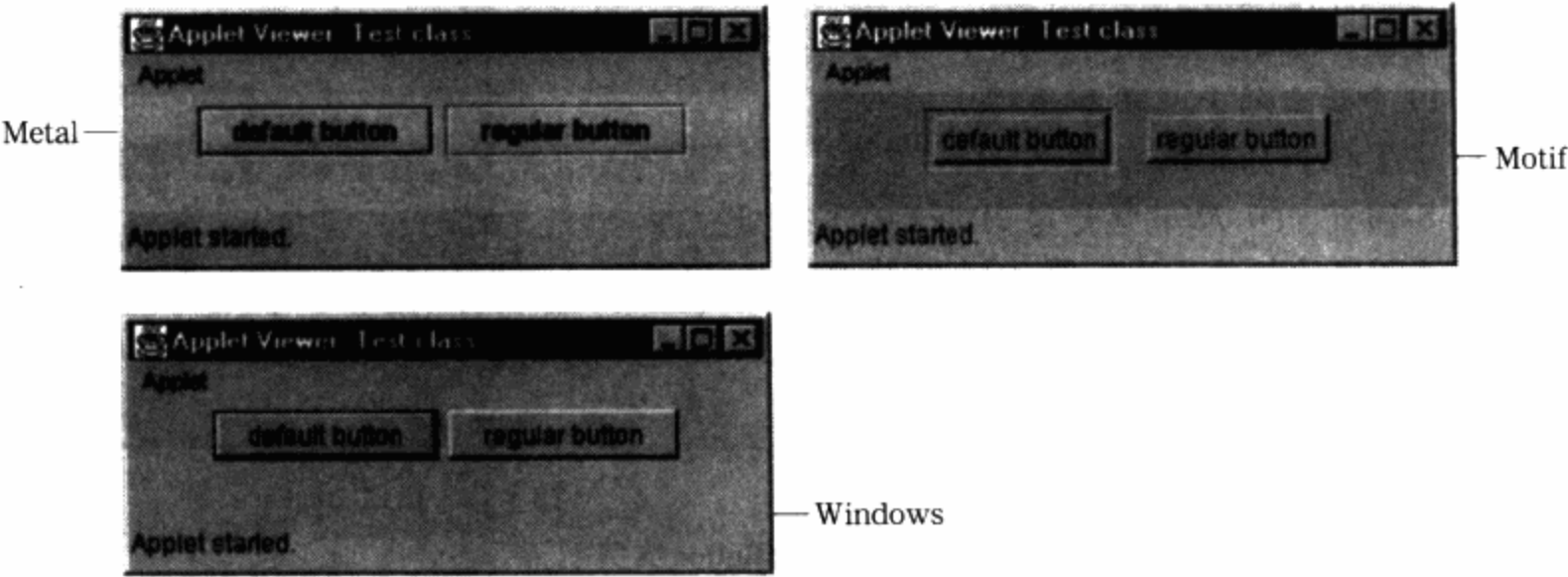


图 8-18 不同界面样式的缺省按钮

图 8-18 中示出的小应用程序包含了两个按钮，其中一个指定为缺省按钮。
例 8-10 中列出了图 8-18 示出的小应用程序的代码。

例 8-10 把一个按钮指定为缺省按钮

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JRootPane rootPane = getRootPane ();
        JButton def = new JButton ("default button");
        JButton reg = new JButton ("regular button");
        rootPane.setDefaultButton (def);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (def);
        contentPane.add (reg);
    }
}
```

AbstractButton 几乎提供了 JButton 中实现的所有功能，并为反复按钮和单选按钮、复选框、菜单和菜单项提供了基础。类总结 8-4 中对 AbstractButton 类进行了总结。

类总结 8-4 AbstractButton

扩展: JComponent

实现: Swing.Constants、java.awt.ItemSelectable

1. 构造方法

public AbstractButton ()

AbstractButton 的这个无参数构造方法是由编译器产生的，因为 AbstractButton 不显式地实现一个构造方法。由于 AbstractButton 是一个抽象类，它的无参数构造方法将仅从扩展类的构造方法中调用。

2. 方法

(1) 检验

protected int checkHorizontalKey (int, String)

protected int checkVerticalKey (int, String)

上面列出的两个方法对传送给它们的整型值进行检验，确保它们是指定按钮内容的水平和垂直排列方式及文本相对于图标的位置的有效值。有效值列表参见表 8-2。如果这些值是无效的，这两个方法都弹出一个 IllegalArgumentException 异常信息，并且把这两个方法所载入的字符串用作这个异常产生的消息。如果这些值是有效值，则这两个方法无反应地返回。

这些方法是 protected 方法，因此，JButton 的扩展能够重载它们，但是，实践中很少重载它们。

(2) 动作与子项监听器

public void addActionListener (ActionListener)

public void addChangeListener (ChangeListener)

public void removeActionListener (ActionListener)

public void removeChangeListener (ChangeListener)

protected ActionListener createActionListener ()

protected ChangeListener createChangeListener ()

protected ItemListener createItemListener ()

protected void fireActionPerformed (ActionEvent)

protected void fireItemStateChanged (ItemEvent)

protected void fireStateChanged ()

public Object [] getSelectedObjects ()

public void addItemListener (ItemListener)

public void removeItemListener (ItemListener)

上面列出的前两组 public 方法使得可以在 Swing 按钮上添加和删除动作监听器和变化监听器。

上面列出的 protected 方法完成两个功能。其中的 create... 方法返回的监听器可以添加到按钮的模型中。缺省时，这些 create... 方法是这样来实现的，它们返回的监听器简单地向按钮的监听器传递模型事件。在一些需要以不同的方式处理模型事件的很特殊情况下，则需要重载这些方法，以便返回定制的监听器。

上面列出的 fire... 方法向相应类型的监听器激发事件，这些监听器是已向按钮登记过的。同样，如果缺省行为是不可接受的，则这些方法可以被 AbstractButton 的扩展重载。

`getSelectedObjects` 方法是 `ItemSelectable` 接口的一部分，用于返回一个 `Object` 数组，这个数组包含对按钮的已选取对象的引用。不是所有的 `AbstractButton` 扩展都有可选取子项，因此，`AbstractButton` 实现返回一个 `null` 引用的 `getSelectedObjects` 方法。

(3) 初始化/边框

```
protected void init (String, Icon)
protected void paintBorder (Graphics)
```

这个 `init` 方法设置布局管理器，设置文本和图标，并更新 UI（通过调用 `updateUI()`）。它还添加一个焦点监听器，这个监听器在获得或失去焦点时及在按钮的排列方式被设置时为可访问性目的激发属性变化事件。虽然 `AbstractButton.init()` 是一个 `protected` 方法，但开发者应当在扩展类中重载这个 `init` 方法时确保正确地初始化按钮。

为了使其超类（`JComponent`）实现的 `paintBorder()` 方法仅当按钮的边框绘制属性为 `true` 时才被调用，则可以重载 `paintBorder()` 方法。

(4) 在程序中激活

```
public void doClick ()
public void doClick (int)
```

`AbstractButton` 实例可以用上面列出的方法用程序激活。传送给 `doClick()` 的 `integer` 值表示按钮保持按下状态的时间（以毫秒为单位）。无参数的 `doClick()` 调用 `doClick(68)`，即按钮保持按下状态 68 毫秒。

图 8-19 中示出的小应用程序包含两个按钮——当激活左边的按钮时，它为右边的按钮调用 `doClick(int)`。传送给 `doClick()` 的整数值是这个小程序的组合框中显示的当前值。

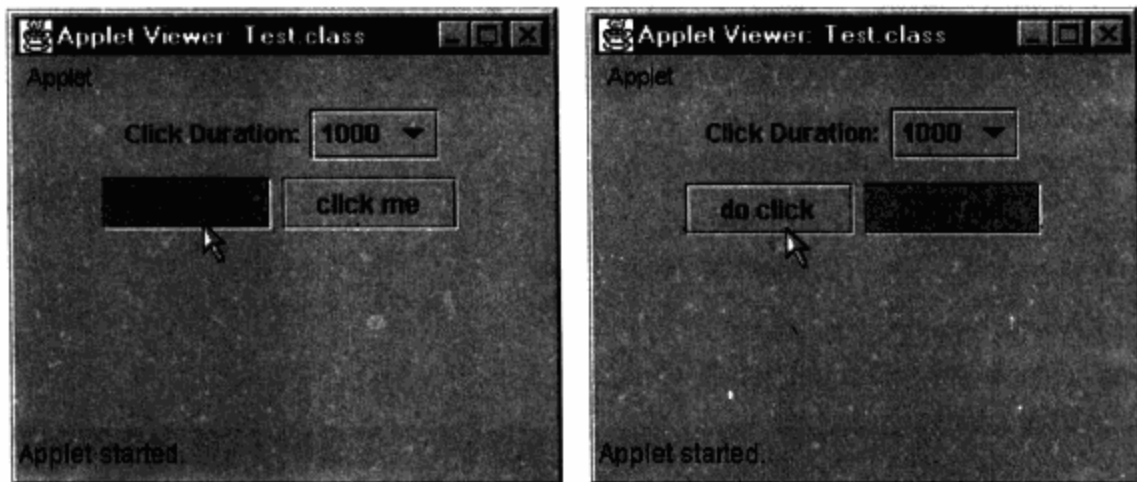


图 8-19 程序方式单击一个按钮

这个小应用程序的内容窗格包含两个面板：一个包含标签和组合框的控制面板和一个包含了两个按钮的面板。“doClick”的动作监听器只简单地为“click me”按钮调用 `doClick()`，而组合框的子项监听器则设置 `clickDuration` 值。

例 8-11 中列出了图 8-19 示出的小应用程序的代码。

例 8-11 程序方式单击一个按钮

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
    int clickDuration = 68;

    public Test () {
```

```

Container contentPane = getContentPane ();
JPanel controlPanel = new JPanel ();
JPanel buttonPanel = new JPanel ();

JButton doClick = new JButton ("do click");
final JButton clickMe = new JButton ("click me");

final JComboBox comboBox = new JComboBox (new Object [] {
    "68", "250", "500", "750", "1000"
});

controlPanel.add (new JLabel ("Click Duration:"));
controlPanel.add (comboBox);

buttonPanel.add (doClick);
buttonPanel.add (clickMe);

contentPane.add (controlPanel, BorderLayout.NORTH);
contentPane.add (buttonPanel, BorderLayout.CENTER);

getRootPane ().setDefaultButton (doClick);

doClick.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        clickMe.doClick (clickDuration);
    }
});

comboBox.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        if (e.getStateChange () == ItemEvent.SELECTED) {
            clickDuration = Integer.parseInt ((String)
                comboBox.getSelectedItem ());
        }
    }
});
}
}

```

(5) 图标

```

public Icon getIcon ()
public Icon getDisabledIcon ()
public Icon getDisabledSelectedIcon ()
public Icon getPressedIcon ()
public Icon getRolloverIcon ()
public Icon getRolloverSelectedIcon ()
public Icon getSelectedIcon ()

public void setIcon (Icon)
public void setDisabledIcon (Icon)
public void setDisabledSelectedIcon (Icon)
public void setPressedIcon (Icon)
public void setRolloverIcon (Icon)
public void setRolloverSelectedIcon (Icon)
public void setSelectedIcon (Icon)

```

AbstractButton 为其支持的所有图标都提供了获取方法和设置方法。表 8-4 中列出了 AbstractButton 支持的图标和它们的用法。JButton 仅使用了 7 种图标中的 4 种：缺省的、鼠标滚过的、按下的和禁用的。JButton 类不使用选取的图标，因为不能选取 JButton 的实例。鼠标滚过的、选取的图标和禁用的、已选取的图标在 Swing 中未被使用。

图 8-20 中示出的小应用程序包含一个 JButton 实例，这个实例能够使用 AbstractButton 类支

持的 7 种图标类型。

图 8-20 中显示的图标是一个 `StringIcon` 类的实例。`StringIcon` 类显示一个在创建时刻带入的字符串。`StringIcon` 类实现 `Icon` 接口：

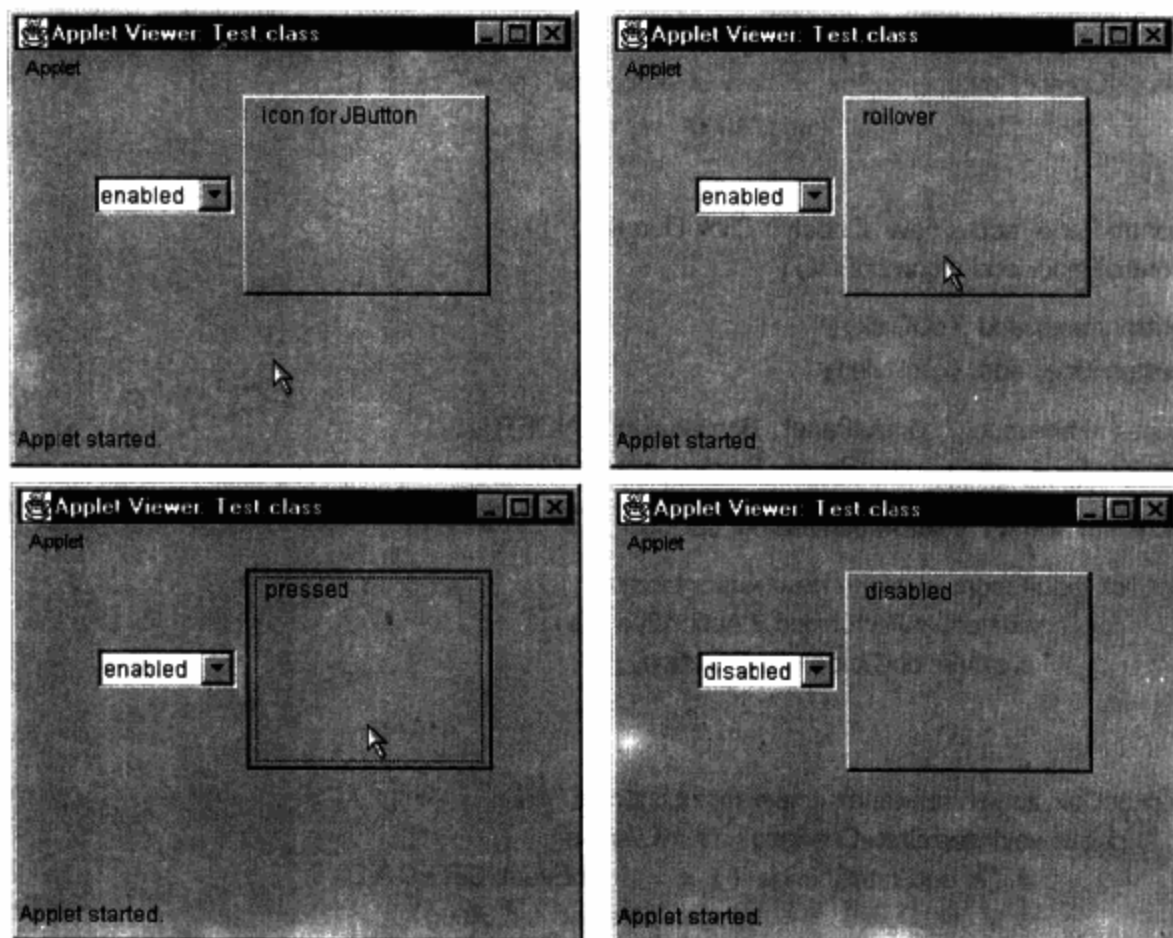


图 8-20 JButton 图标

```
class StringIcon implements Icon {
    private String s;

    public StringIcon (String s) {
        this.s = s;
    }

    public int getIconWidth () { return 100; }
    public int getIconHeight () { return 100; }

    public void paintIcon (Component c, Graphics g, int x, int y) {
        FontMetrics fm = g.getFontMetrics ();
        g.setColor (c.getForeground ());
        g.drawString (s, 10, fm.getHeight ());
    }
}
```

这个小应用程序创建了 `StringIcon` 的七个实例，并调用相应的 `AbstractButton` 方法来分配这些图标。

```
public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        Icon icon = new StringIcon ("icon for JButton"),
            rolloverIcon = new StringIcon ("rollover"),
            pressedIcon = new StringIcon ("pressed"),
            disabledIcon = new StringIcon ("disabled"),
            selectedIcon = new StringIcon ("selected"),
            rolloverSelectedIcon =
                new StringIcon ("rollover selected"),
```



```

        disabledSelectedIcon =
            new ImageIcon ("disabled selected");
        final JButton button = new JButton ();
        button.setRolloverEnabled (true);
        button.setIcon (icon);
        button.setRolloverIcon (rolloverIcon);
        button.setRolloverSelectedIcon (rolloverSelectedIcon);
        button.setSelectedIcon (selectedIcon);
        button.setPressedIcon (pressedIcon);
        button.setDisabledIcon (disabledIcon);
        button.setDisabledSelectedIcon (disabledSelectedIcon);
        ...
    }
    ...
}

```

例 8-12 列出了图 8-20 中示出的小应用程序的代码。

例 8-12 JButton 图标

```

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        Icon icon = new ImageIcon ("icon for JButton"),
            rolloverIcon = new ImageIcon ("rollover"),
            pressedIcon = new ImageIcon ("pressed"),
            disabledIcon = new ImageIcon ("disabled"),
            selectedIcon = new ImageIcon ("selected"),
            rolloverSelectedIcon =
                new ImageIcon ("rollover selected"),
            disabledSelectedIcon =
                new ImageIcon ("disabled selected");

        final JButton button = new JButton ();
        button.setRolloverEnabled (true);
        button.setIcon (icon);
        button.setRolloverIcon (rolloverIcon);
        button.setRolloverSelectedIcon (rolloverSelectedIcon);
        button.setSelectedIcon (selectedIcon);
        button.setPressedIcon (pressedIcon);
        button.setDisabledIcon (disabledIcon);
        button.setDisabledSelectedIcon (disabledSelectedIcon);

        JComboBox cb = new JComboBox ();
        cb.addItem ("enabled");
        cb.addItem ("disabled");

        cb.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                if (e.getStateChange () == ItemEvent.SELECTED) {

```

```

        String item = (String) e.getItem ();
        if (item.equals ("enabled")) {
            button.setEnabled (true);
        }
        else {
            button.setEnabled (false);
        }
    }
}

contentPane.setLayout (new FlowLayout ());
contentPane.add (cb);
contentPane.add (button);
}

class StringIcon implements Icon {
    private String s;

    public StringIcon (String s) {
        this.s = s;
    }

    public int getIconWidth () { return 100; }
    public int getIconHeight () { return 100; }

    public void paintIcon (Component c, Graphics g, int x, int y) {
        FontMetrics fm = g.getFontMetrics ();
        g.setColor (c.getForeground ());
        g.drawString (s, 10, fm.getHeight ());
    }
}

```

(6) 布尔属性访问方法

```

public boolean isBorderPainted ()
public boolean isContentAreaFilled ()
public boolean isFocusPainted ()
public boolean isRolloverEnabled ()
public boolean isSelected ()

public void setBorderPainted (boolean)
public void setContentAreaFilled (boolean)
public void setFocusPainted (boolean)
public void setRolloverEnabled (boolean)
public void setSelected (boolean)

```

上面列出的 `AbstractButton` 的布尔属性的访问方法控制按钮是否绘制边框、是否绘制一个焦点指示器、是否显示光标进入图标，以及按钮当前是否已选取。

对透明按钮应当用 `setContentAreaFilled` 方法替代 `setOpaque` 方法。按钮是否遵守重画管理器的不透明概念是与界面样式有关的，因此，应当使用 `set...` 方法，而不用 `setOpaque ()` 方法。

(7) 属性访问方法

```

public String getActionCommand ()
public int getHorizontalAlignment ()
public int getHorizontalTextPosition ()
public int getMnemonic ()
public ButtonModel getModel ()

```

```

public Insets getMargin ()
public String getText ()
public ButtonUI getUI ()
public int getVerticalAlignment ()
public int getVerticalTextPosition ()

public void setActionCommand ()
public void setEnabled (boolean)
public void setHorizontalAlignment (int)
public void setHorizontalTextPosition (int)
public void setMargin (Insets)
public void setMnemonic (char)
public void setModel (ButtonModel)
public void setText (String)
public void setUI (ButtonUI)
public void setVerticalAlignment (int)
public void setVerticalTextPosition (int)

```

上面列出的方法是访问 `AbstractButton` 的属性的方法。文本、图标、排列方式和文本位置的访问方法与 `JLabel` 类的相应访问方法具有完全相同的原型 (signature)。 `parainString` 方法返回一个抽象按钮的字符串表示。

1. 内容排列方式与文本位置

图 8-21 中示出的两个小应用程序与图 8-4 及图 8-5 中示出的小应用程序几乎相同。不同之处在于，图 8-21 操纵 `JButton` 实例的排列方式和文本位置，而不是操纵 `JLabel` 的排列方式和文本位置。由于 `AbstractButton` 和 `JLabel` 访问通用属性的 API 几乎相同，这个小应用程序的 `JLabel` 与 `JButton` 版本的唯一区别仅仅是所创建的组件的类型不同。因此，这个小应用程序的 `JButton` 版本在书中未列出，但它们包含在随书所带的 CD 盘中。

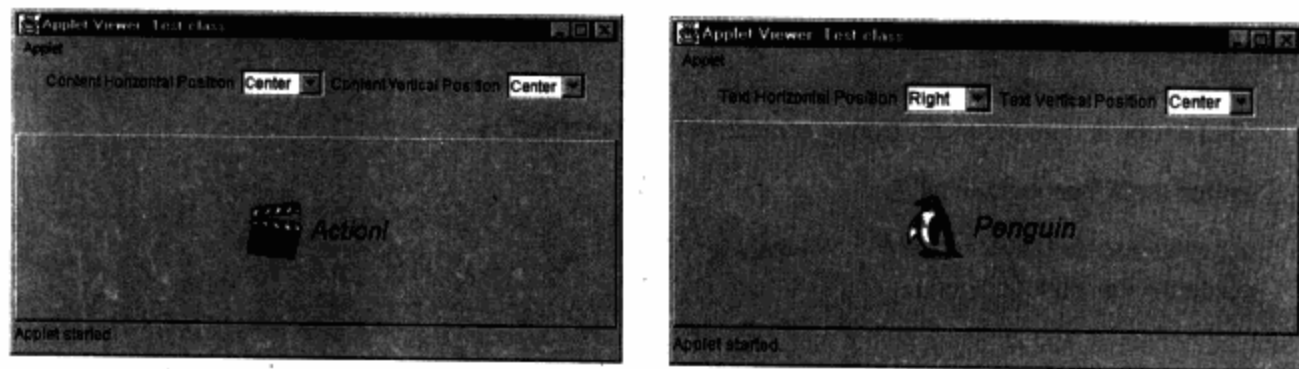


图 8-21 Swing 按钮的内容排列和文章位置

2. 按钮边距

`AbstractButton` 维护一个 `margin` 属性，它指从按钮边框的内边沿到按钮的内容的外边沿的距离。图 8-22 中示出的小应用程序把按钮的边距设置为一个 `Insets (50, 25, 10, 5)` 值[⊖]。例 8-13 中列出了图 8-22 示出的小应用程序代码。

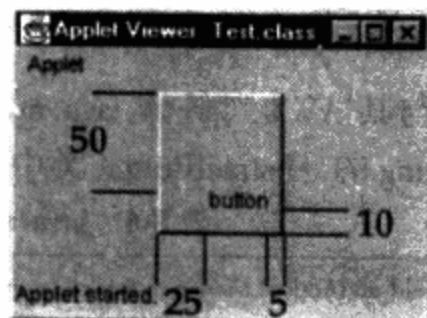


图 8-22 按钮边距

⊖ `Insets` 构造方法的参数是：上、左、下和右边距。

例 8-13 设置按钮边距

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane () ;
        JButton button = new JButton ("button") ;

        button.setMargin (new Insets (50, 25, 10, 5)) ;
        contentPane.setLayout (new FlowLayout ()) ;
        contentPane.add (button) ;
    }
}
```

3. 按钮助记符

图 8-23 中示出的小应用程序为它所显示的按钮设置了一个助记符。一个按钮助记符的视觉指示标志 (visual indicator)，在不同的界面样式中可以是不同；但是，通常按钮上的助记符是带下划线的，并且同时按下变位 (Meta) 符和助记符将激活该按钮[⊖]。当按钮具有焦点时按下助记符将激活该按钮。

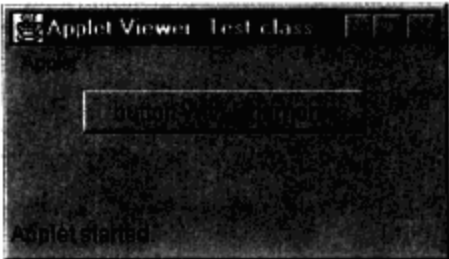


图 8-23 按钮助记符

例 8-14 列出了图 8-23 中示出的小应用程序的代码。

例 8-14 按钮助记符

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane () ;
        JButton button = new JButton ("button With Mnemonic") ;

        button.setMnemonic ('M') ;

        contentPane.setLayout (new FlowLayout ()) ;
        contentPane.add (button) ;
    }
}
```

8.4.4 AWT 兼容

JButton 与其 AWT 对等体 java.awt.Button 是源代码兼容的。Java.awt.Button 类的所有公共方法都能在 Swing 的 AbstractButton 类中找到。表 8-6 中列出了这两个类所共有的方法。

表 8-6 java.awt.Button 和 AbstractButton 共有的方法

public void addActionListener (ActionListener) ;
public void removeActionListener (ActionListener) ;

⊖ 变位符与平台有关；例如，Windows 的变位符是 Alt 键。

(续)

```
public String getLabel ();  
public String getActionCommand ();  
public void setActionCommand (String);  
public void setLabel (String);
```

8.5 本章回顾

Swing 标签和按钮（分别由 `JLabel` 和 `JButton` 类代表）是一对奇怪的伙伴。标签和按钮都能够显示文本或图标。因此，`JLabel` 类中的 20 个方法在 `JButton` 类中有功能对等的方法；但是，`JLabel` 与 `JButton` 之间不是通过继承关系联系在一起的。

遗憾的是，当两个类有大量相同的功能，但不是通过继承关系联系起来的时候，则能够以完全相同的方式操纵它们的唯一保证是程序员的素养。换言之，`JLabel` 和 `JButton` 共有属性的访问方法应当具有原型完全相同的获取方法和设置方法。由于这些方法不是继承的，所以确保一致性是开发人员，而不是编译器的责任。`JLabel` 和 `JButton` 对共有的属性具有统一的访问方法，唯一的例外是助记符属性，对标签该属性是通过 `getDisplayMnemonic()` 访问的，而对按钮则是通过 `getMnemonic()` 访问的。

`JLabel` 和 `AbstractButton` 不扩展封装它们的共有功能的基类。这个事实导致了这两个类之间的其他一些不一致性。例如，一个标签显示的文本和图标之间的间隙是可设置的，但对按钮却不是如此。另外，虽然按钮和标签都可以有禁用图标，但只有按钮可以有“滚过式”图标，这个图标是在光标进入按钮时显示的。

另一方面，Swing 按钮的几乎所有功能都是封装在 `AbstractButton` 类中的，这样，`AbstractButton` 类的所有扩展（包括菜单和菜单项）都能以一种一致的方式操作。这比 AWT 是一个巨大进步。AWT 没有为按钮提供一个公共基类，没有把菜单和菜单项作为按钮类的扩展（甚至没有作为 AWT 组件类的扩展）来实现。

第9章 反转按钮、复选框和单选钮

本章介绍三种 Swing 按钮：反转按钮、复选框和单选钮，它们分别由 `JToggleButton`、`JCheckBox` 和 `JRadioButton` 类代表。

这三种按钮都是 `AbstractButton` 类的最终扩展。`AbstractButton` 类实现了这三个类提供的几乎所有功能。有关 `AbstractButton` 类的更多信息，请参见第8章“标签与按钮”。

9.1 `JToggleButton` 类

反转按钮是具有“选取”和“取消选取”两个状态的按钮。`JToggleButton` 类是 Swing 复选框和 Swing 单选钮的超类。

除了作为复选框和单选钮的基类外，`JToggleButton` 组件还有其自身的用途。图9-1示出的小应用程序带有一个 `JToggleButton` 实例。左图中的按钮处于“未选取”状态，而右图中显示的是已选取的按钮。

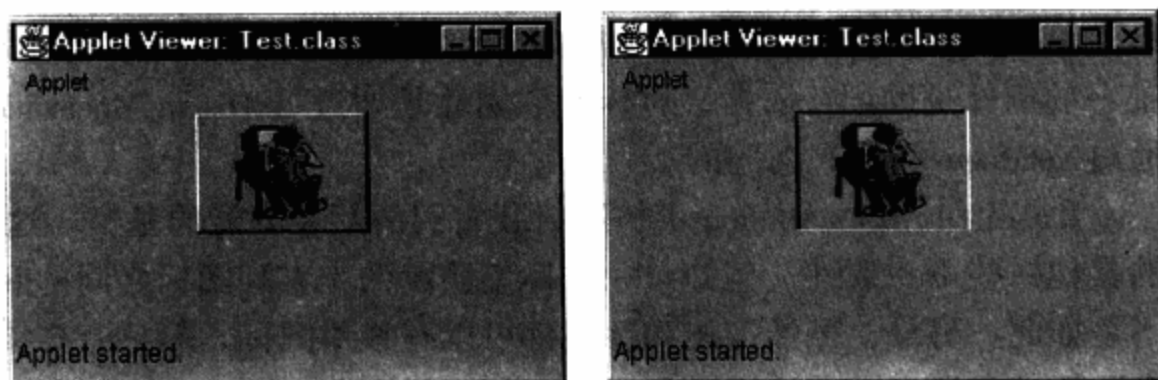


图 9-1 反转按钮

例 9-1 中列出了图9-1示出的小应用程序的代码。

例 9-1 `JToggleButton` 的一个简单例子

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane () ;
        ImageIcon icon = new ImageIcon ("togglebuttonImage.gif") ;
        JToggleButton button = new JToggleButton (icon) ;

        contentPane.setLayout (new FlowLayout ()) ;
        contentPane.add (button) ;
    }
}
```

这个小应用程序创建了一个图像图标和一个 `JToggleButton` 实例。这个小应用程序的内容窗格的布局管理器是一个 `FlowLayout` 实例，而这个按钮是放在该小应用程序的内容窗格中的。

组件总结 9-1 中对 `JToggleButton` 类作了总结。

组件总结 9-1 `JToggleButton`

模型： `JToggleButton.ToggleButtonModel`

UI 代表: javax.swing.plaf.basic.Basic ToggleButtonUI
 绘制器: ——
 编辑器: ——
 激发的事件: ActionEvent/ChangeEvent/ItemEvent
 替代: ——
 类图: 参见图 8-12

9.1.1 JToggleButton 属性

JToggleButton 类没有自己的属性——所有的属性都是从 AbstractButton 类继承而来的。有关 AbstractButton 类属性的更多信息, 请参见 8.4.2 节“JButton 事件”。

9.1.2 JToggleButton 事件

与所有的 Swing 按钮一样, 当激活反转按钮时, 它们激发动作事件, 当修饰它们的关联属性时激发属性改变事件, 当它们的状态改变时激发变化事件。按钮激发的动作事件和变化事件的更多信息, 请参见 8.4.2 节“JButton 事件”, 有关属性变化事件的更多信息则参见属性变化事件。另外, 当选取或取消选取反转按钮时, 它们激发子项事件。ItemListener 接口和 ItemEvent 类在 java.awt.event 包中。接口总结 9-1 中对 ItemListener 接口作了总结。

接口总结 9-1 ItemListener

```
public abstract void itemStateChanged (ItemEvent)
```

ItemListener.itemStateChanged() 载入了一个 ItemEvent 实例。类总结 9-1 中对 ItemEvent 类进行了总结。

类总结 9-1 ItemEvent

扩展: java.awt.AWTEvent

1. 常量

```
public static final int ITEM_FIRST
public static final int ITEM_LAST
public static final int ITEM_STATE_CHANGED
public static final int SELECTED
public static final int DESELECTED
```

上面所列的第一组常量表示动作事件的有效事件 ID。有关事件 ID 的更多信息, 请参见《Java2 图形设计, 卷 I: AWT》1。SELECTED 和 DESELECTED 常量表示子项事件代表一个子项是选取还是取消选取。

2. 构造方法

```
public ItemEvent (ItemSelectable source, int id, Object item, int stateChange)
```

子项事件是用事件源、事件 ID、子项和状态变化创建的。状态变化总是 ItemEvent.SELECTED 或 ItemEvent.DESELECTED。

3. 方法

```
public Object getItem ()
public ItemSelectable getItemSelectable ()
public int getStateChange ()
```



```
public String paramString ()
```

ItemEvent 类为获得对子项、事件源和对事件表示的状态变化的引用提供了上述方法。还提供了一个 paramString 方法来创建从 ItemEvent.toString() 返回的字符串。

图 9-2 中示出的小应用程序包含了一个配备了一个子项监听器的反转按钮。这个监听器通过更新该小应用程序的状态区来反应按钮的选取和取消选取。例 9-2 中列出了图 9-2 中示出的小应用程序的代码。

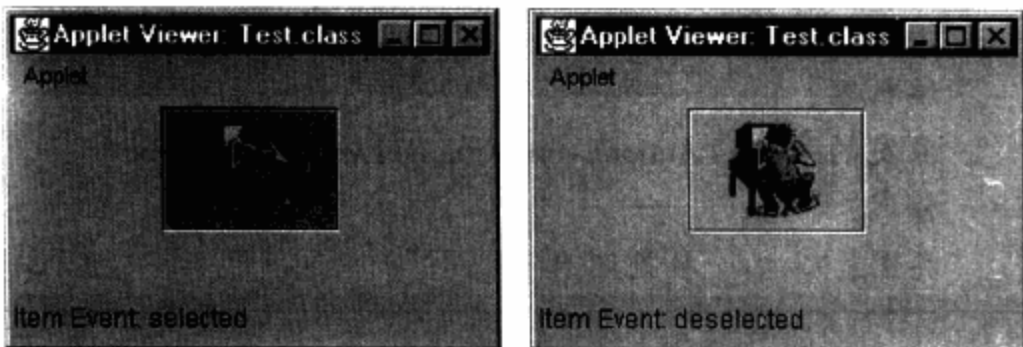


图 9-2 处理 JToggleButton 的选取和取消选取

例 9-2 用一个子项监听器来处理反转按钮的选取操作

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        ImageIcon icon = new ImageIcon ("togglebuttonImage.gif");
        final JToggleButton button = new JToggleButton (icon);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                int state = e.getStateChange ();
                String s;

                if (state == ItemEvent.SELECTED) s = "selected";
                else s = "deselected";

                showStatus ("Item Event: " + s);
            }
        });
    }
}
```

这个子项监听器确定是否是选取并更新小应用程序的状态区。

9.1.3 JToggleButton 类总结

类总结 9-2 中列出了 JToggleButton 的 public 和 protected 变量和方法。

类总结 9-2 JToggleButton

扩展: JComponent

实现: java.accessibility.Accessible

1. 构造方法

```
public JToggleButton ()
```

```

public JToggleButton (Icon)
public JToggleButton (Icon, boolean selected)
public JToggleButton (String)
public JToggleButton (String, boolean selected)
public JToggleButton (String, Icon)
public JToggleButton (String, Icon, boolean selected)

```

JToggleButton 类为创建各种不同配置的反转按钮提供了七种构造方法。传送给这些构造方法的 boolean 值指定初始化时按钮是否被选取——值为 true 表示选取。

图 9-3 示出了一个创建七个 JToggleButton 实例的小应用程序

这个小应用程序用上面列出的 JToggleButton 构造方法创建了多个按钮。带 boolean 参数的构造方法是用 true 值调用的。以指定按钮最初是被选取的。

例 9-3 中列出了图 9-3 中示出的小应用程序的代码。

例 9-3 创建反转按钮

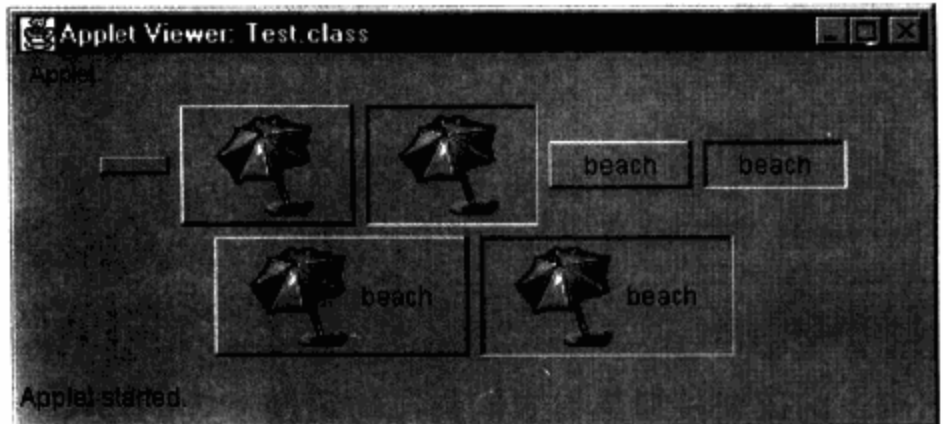


图 9-3 创建 JToggleButton 实例

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
public Test () {
    Container contentPane = getContentPane ();
    Icon icon = new ImageIcon ("beach_umbrella.gif");
    JToggleButton button_1 = new JToggleButton (),
        button_2 = new JToggleButton (icon),
        button_3 = new JToggleButton (icon, true),
        button_4 = new JToggleButton ("beach"),
        button_5 = new JToggleButton ("beach", true),
        button_6 = new JToggleButton ("beach", icon),
        button_7 = new JToggleButton ("beach", icon,
                                    true);

    contentPane.setLayout (new FlowLayout ());
    contentPane.add (button_1);
    contentPane.add (button_2);
    contentPane.add (button_3);
    contentPane.add (button_4);
    contentPane.add (button_5);
    contentPane.add (button_6);
    contentPane.add (button_7);
}
}

```

2. 方法

```

public String paramString ()
public AccessibleContext getAccessibleContext ()

```

```
public String getUIClassID ()
public void updateUI ()
```

paramString 方法重载 AbstractButton 类中的 paramString 方法，它创建一个字符串，提供与反转按钮有关的信息。上面列出的最后三个方法可以在 JComponent 的大多数扩展中找到。Swing 轻量组件能够返回与可访问性有关的内容，包括组件的可访问性信息和它们的 UI 代表的类名。当组件装配了一个 UI 代表时，则调用上面的 updateUI 方法。

9.1.4 AWT 兼容

由于 AWT 没有提供类似的组件，因此，JToggleButton 类不与任何 AWT 组件兼容。

Swing 提示

Swing 的反转按钮具有它们自己的用途

虽然 JToggleButton 类的主要作用是作为 JCheckBox 和 JRadioButton 的基类，但 Swing 的反转按钮有它们自己的作用。JToggleButton 类是一个具体（非抽象）类，因此，与其他 Swing 按钮一样，Swing 反转按钮能够被实例化并添加到容器中。Swing 反转按钮的一种用途在图 9-4 中进行了描述。通过提供一组相互排斥的按钮，反转按钮可以有与单选钮类似的使用方法。

9.2 按钮组

通过使用 ButtonGroup 类，Swing 允许一组按钮具有相互排斥的选取行为。创建按钮并把它们添加到一个按钮组中，这样，在这个按钮组中选取一个按钮使得前一个被选取按钮取消选取。

图 9-4 中示出的小应用程序示出了一种相当常见的用户界面技术——一组能够以相互排斥的方式选取的按钮。

这个小应用程序使用了一个容器，该容器是 Box 类的一个实例，用来把按钮排列在一列中。当按钮添加到这个框（Box）中时，它们还添加到一个按钮组中。

例 9-4 中列出了图 9-4 中示出的小应用程序的原代码。

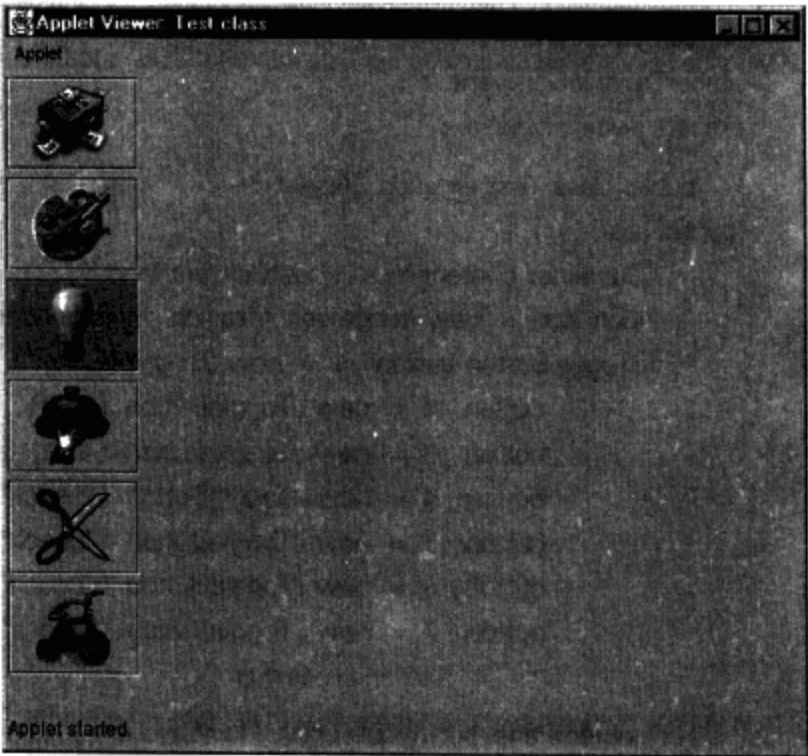


图 9-4 一个按钮组中的反复按钮

例 9-4 为相互排斥的选取行为使用一个按钮组

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        ButtonGroup group = new ButtonGroup ();
```

```

JToggleButton [] buttons = new JToggleButton [] {
    new JToggleButton (new ImageIcon ("ballot _ box.gif")),
    new JToggleButton (new ImageIcon ("palette.gif")),
    new JToggleButton (new ImageIcon ("light _ bulb1.gif")),
    new JToggleButton (new ImageIcon ("dining.gif")),
    new JToggleButton (new ImageIcon ("scissors.gif")),
    new JToggleButton (new ImageIcon ("tricycle.gif")),
};
Box box = Box.createVerticalBox ();
for (int i=0; i < buttons.length; ++i) {
    group.add (buttons [i]);
    box.add (Box.createVerticalStrut (5));
    box.add (buttons [i]);
}
box.add (Box.createVerticalStrut (5));
contentPane.add (box);

```

注意，这个按钮组未添加到容器中，因为 `ButtonGroup` 的实例不是组件。
类总结 9-3 中列出了 `ButtonGroup` 类的 `public` 和 `protected` 方法。

类总结 9-3 ButtonGroup

扩展：`java.lang.Object`

实现：`java.io.Serializable`

1. 构造方法

`public ButtonGroup ()`

这个无参数构造方法是 `ButtonGroup` 类提供的唯一一个构造方法。

2. 方法

`public void add (AbstractButton)`

`public void remove (AbstractButton)`

`public Enumeration getElement ()`

`public ButtonModel getSelection ()`

`public void setSelected (ButtonModel, boolean)`

`public boolean isSelected (ButtonModel)`

上面列出的第一组方法用于从按钮组中添加和删除按钮，并用于获得对组中现有按钮的枚举。`getSelection` 方法返回被选取按钮的模型。最初，`getSelection` 方法返回 `null`，直到组中有一个按钮被选取。`getSelection` 方法既可用于选取组中的一个按钮，也可用于取消选取组中的一个按钮，这取决于 `boolean` 参数的值。`isSelected` 方法可用于确定组中的一个按钮是否被选取。

9.3 复选框

复选框也是具有两种状态的按钮。它们通常显示文本，并有一个指示它们当前是否被选取的可视指示器。复选框通常显示在组中，但通常禁用相互排斥的选取行为。

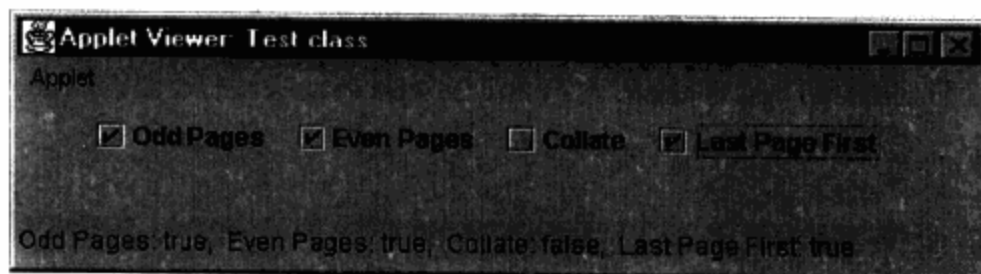


图 9-5 复选框

图 9-5 示出的小应用程序包含四个用来选取打印选项的复选框。这四个复选框放在一个组中。不论何时选取或取消选取了一个复选框，这个小应用程序都用每个复选框的当前状态更新它的状态栏。

这个小应用程序创建了一个 `PrintOptionsPanel` 实例，并把它添加到内容窗格中。

```
public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        contentPane.add (new PrintOptionsPanel (this), BorderLayout.CENTER);
    }
}
```

这个 `PrintOptionsPanel` 是一个包含复选框的 `JPanel` 扩展。`PrintOptionsPanel` 的构造方法对以这个小应用程序的一个引用作为参数，以便能够更新这个小应用程序的状态条。这个构造方法创建了一些复选框，并把它们添加到面板中。

```
class PrintOptionsPanel extends JPanel {
    JCheckBox oddPages, evenPages, collate, lastFirst;
    Listener listener = new Listener ();
    JApplet applet;

    public PrintOptionsPanel (JApplet applet) {
        this.applet = applet;
        oddPages = new JCheckBox ("Odd Pages");
        evenPages = new JCheckBox ("Even Pages");
        collate = new JCheckBox ("Collate");
        lastFirst = new JCheckBox ("Last Page First");

        oddPages.addItemListener (listener);
        evenPages.addItemListener (listener);
        collate.addItemListener (listener);
        lastFirst.addItemListener (listener);

        add (oddPages);
        add (evenPages);
        add (collate);
        add (lastFirst);
    }
}
```

当它们被选取或取消选取时，`JCheckBox` 的实例都激发子项事件。`PrintOptionsClass` 实现更新该小应用程序状态栏的子项监听器。

```
...
class Listener implements ItemListener {
    public void itemStateChanged (ItemEvent event) {
        applet.showStatus (
            "Odd Pages: " + oddPages.isSelected () + ", " +
            "Even Pages: " + evenPages.isSelected () + ", " +
            "Collate: " + collate.isSelected () + ", " +
            "Last Page First: " + lastFirst.isSelected ());
    }
}
...
```

例 9-5 中完整地列出了图 9-5 所示的小应用程序的代码。

例 9-5 运行中的 Swing 复选框

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        contentPane.add (new PrintOptionsPanel (this), BorderLayout, CENTER);
    }
}

class PrintOptionsPanel extends JPanel {
    JCheckBox oddPages, evenPages, collate, lastFirst;
    Listener listener = new Listener ();
    JApplet applet;

    public PrintOptionsPanel (JApplet applet) {
        this.applet = applet;
        oddPages = new JCheckBox ("Odd Pages");
        evenPages = new JCheckBox ("Even Pages");
        collate = new JCheckBox ("Collate");
        lastFirst = new JCheckBox ("Last Page First");

        oddPages.addItemListener (listener);
        evenPages.addItemListener (listener);
        collate.addItemListener (listener);
        lastFirst.addItemListener (listener);

        add (oddPages);
        add (evenPages);
        add (collate);
        add (lastFirst);
    }
}

class Listener implements ItemListener {
    public void itemStateChanged (ItemEvent event) {
        applet.showStatus (
            "Odd Pages: " + oddPages.isSelected () + ", " +
            "Even Pages: " + evenPages.isSelected () + ", " +
            "Collate: " + collate.isSelected () + ", " +
            "Last Page First: " + lastFirst.isSelected ();
        );
    }
}

```

组件总结 9-2 中对 JCheckBox 类进行了总结。

组件总结 9-2 JCheckBox

模型:	JToggleButton.ToggleButtonModel
UI 代表:	javax.swing.plaf.basic.BasicCheckBoxUI
绘制器:	——
编辑器:	——
激发的事件:	ActionEvent/ChangeEvent/ItemEvent

替代: java.awt.CheckBox
类图: 参见图 8-12

9.3.1 JCheckBox 属性

JCheckBox 没有自己的属性——所有属性都是从 AbstractButton 类继承来的。

9.3.2 JCheckBox 事件

与所有的 Swing 按钮一样，复选框在激活时激发动作事件，当关联属性修改时激发属性变化事件，并在它们的状态改变时激发变化事件。有关按钮激发的动作事件和变化事件的更多信息参见 8.4.2 节“JButton 事件”，有关属性变化事件的信息则参见 3.2.4 节。另外，当它们被选取时，复选框激发子项事件。用一个子项监听器处理复选框选取和取消的例子参见例9-5。

9.3.3 JCheckBox 类总结

类总结 9-4 中对 JCheckBox 进行了总结。

类总结 9-4 JCheckBox

扩展: JToggleButton
实现: javax.accessibility.Accessible

1. 构造方法

```
public JCheckBox ()  
public JCheckBox (Icon)  
public JCheckBox (Icon, boolean selected)  
public JCheckBox (String)  
public JCheckBox (String, boolean selected)  
public JCheckBox (String, Icon)  
public JCheckBox (String, Icon, boolean selected)
```

JCheckBox 提供的一组构造方法与 JToggleButton 和 JRadioButton 类提供的构造方法相同；JCheckBox 的实例可以用文本和图标的任意组合创建。另外，可以在创建时刻指定复选框的选取状态。

图 9-6 示出了一个创建了 7 个具有不同配置的小应用程序。

例 9-6 列出了图 9-6 中示出的小应用程序的代码。

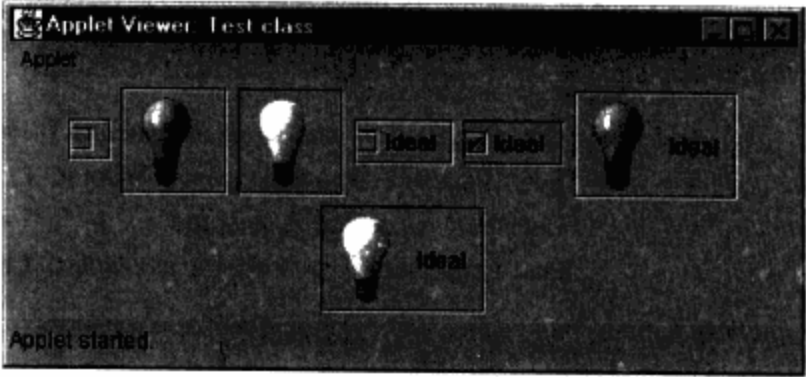


图 9-6 实例化复选框

例 9-6 实例化复选框

```
import java.awt.* ;  
import java.awt.event.* ;  
import javax.swing.* ;  
  
public class Test extends JApplet {  
    public Test () {  
        Container contentPane = getContentPane () ;  
        Icon icon = new ImageIcon ("bulb.gif") ;
```



```

JCheckBox [] checkboxes = new JCheckBox [] {
    new JCheckBox (),
    new JCheckBox (icon),
    new JCheckBox (icon, true),
    new JCheckBox ("idea!"),
    new JCheckBox ("idea!", true),
    new JCheckBox ("idea!", icon),
    new JCheckBox ("idea!", icon, true)
};
contentPane.setLayout (new FlowLayout ());
for (int i=0; i < checkboxes.length; ++i) {
    checkboxes [i] .setBorderPainted (true);
    contentPane.add (checkboxes [i]);
    if (checkboxes [i] .getIcon () != null) {
        checkboxes [i] .setSelectedIcon (
            new ImageIcon ("bulb_bright.gif"));
    }
}
}
}

```

缺省时，复选框不绘制边框。上面列出的小应用程序显式地绘制了每一个复选框的边框，以便能够显示出每个复选框从何处开始及在何处结束。

如果创建一个 JCheckBox 实例时不带图标，则这个复选框上将添加一个小控件来指示该复选框是否被选取。如果为 JCheckBox 实例指定了一个图像，则这个复选框将配备选定的图像来指示复选框的选取状态，如此则不必在复选框上添加控件^①。因此，这个小应用程序为每个具有一个 null 图像的复选框设置选定的图像。

2. 方法

```

public String paramString ()
public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()

```

paramString 方法重载 AbstractButton 类中的 paramString 方法，它创建一个字符串，提供与复选框有关的信息。上面列出的最后三个方法可以在 JComponent 的大多数扩展中找到。Swing 轻量组件能够返回与可访问性有关的内容，包括组件的可访问性信息和它们的 UI 代表的类名。当组件装配了一个 UI 代表时，则调用上面的 updateUI 方法。

Swing 提示

为带有一个图像的复选框显式地设置选定的图标

如果一个 Swing 复选框带有一个图像，则在绘制这个复选框时不绘制指示该复选框是否被选取的控件。因此，如果一个带图像的复选框被选取，则没有视觉指示器来指示选取状态。这样，最好是显式地为一个带图像的复选框设置一个选定的图标。

^① 这种行为对所有标准界面样式都是通用的。

这个建议同样适用于 Swing 单选钮。

AWT 兼容

JCheckBox 与 java.awt.Checkbox 几乎是源代码兼容的。JCheckBox 提供了 getLabel 方法和 setLabel 方法。JCheckBox 和 java.awt.Checkbox 在它们选取时都激发子项事件。JCheckBox 与 java.awt.Checkbox 的 API 在两个方面有区别。首先，只有 AWT 复选框能够添加到 java.awt.CheckGroup 实例中，而任何 Swing 按钮都能添加到 Swing.Buttongroup 实例中。其次，AWT 复选框是通过调用带有一个 boolean 参数值的 getState 方法来选取的，而另一方面，Swing 复选框是通过调用 setSelected 方法来选取的。注意，Swing 实现的复选框比 AWT 具有更好的兼容性。

框几乎是相同的，不同之处仅在于用于表示选取状态所显示的控件。另外，于显示一组相互排斥的选项，即它们通常是放在一个按钮组中的，而复选框斥的选项。

小应用程序包含两个单选
可用于选取要打印的页范
应用程序中的两个单选钮
选取其中之一时，另一个
选取“Print All”按钮时，
本域；选取“Print Range”
文本域，以使用户能够指

程序包含了一个 Print-
它带入了两个参数，即要
束页。

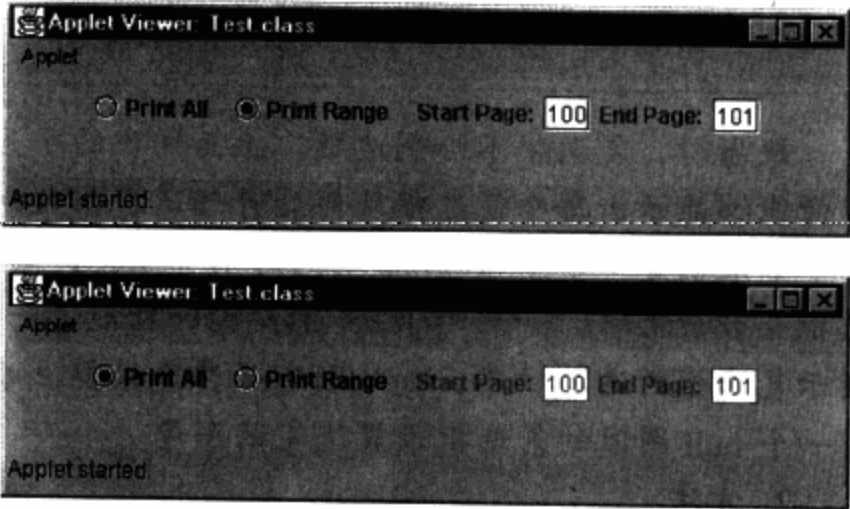


图 9-7 单选钮

```
extends JApplet {  
    ) {  
        contentPane = getContentPane ();  
        add (new PrintRangePanel (100, 101));  
    }
```

是 JPanel 的一个扩展，包含单选钮、标签和文本域。PrintRangePanel 的构造
件，还创建了一个 ButtonGroup 实例，顺序地把两个单选钮加入到这个实例

```
el extends JPanel {  
    ntAll, printRange;  
    , endPage;  
    eld, endField;  
Panel (int start, int end) {  
    group = new ButtonGroup ();  
    w JRadioButton ("Print All");  
    new JRadioButton ("Print Range");
```

9.4 单选钮

单选钮和复选
单选钮几乎总是用
通常用于相互不排
图 9-7 示出的
钮，这两个单选钮
围。包含在这个小
是相互排斥的；当
就取消选取。当选
就禁用标签和文本
按钮则启用标签和
定打印页的范围。

这个小应用
RangePanel 实例，它
打印的开始页和结

```
public class Test ex  
    public void init (  
        Container co  
        contentPane
```

PrintRangePanel
方法实例化这些组
中。

```
class PrintRangePar  
    JRadioButton pri  
    JLabel startPage  
    JTextField startF  
    public PrintRange  
        ButtonGroup  
        printAll = ne  
        printRange =
```

```

...
add (printAll) add (printRange);
add (startPage) add (startField);
add (endPage) add (endField);

printRange.setSelected (true);

group.add (printAll);
group.add (printRange);

```

```

...

```

在这两个单选钮中都添加了子项监听器，以便启用和禁用标签和文本域。

```

...
printRange.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        if (printRange.isSelected ()) {
            startField.setEnabled (true);
            endField.setEnabled (true);
            startPage.setEnabled (true);
            endPage.setEnabled (true);

            startPage.repaint ();
            endPage.repaint ();

            startField.requestFocus ();
        }
    }
});

printAll.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        if (printAll.isSelected ()) {
            startField.setEnabled (false);
            endField.setEnabled (false);
            startPage.setEnabled (false);
            endPage.setEnabled (false);

            startPage.repaint ();
            endPage.repaint ();
        }
    }
});

```

注意：由于，Swing1.1 FCS 中的一个错误，在启用或禁用标签后必须重新绘制它们。例 9-7 中列出了图 9-7 中示出的小应用程序的代码。

例 9-7 运行中的单选钮

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        contentPane.add (new PrintRangePanel (100, 101));
    }
}

```

```

class PrintRangePanel extends JPanel {
    JRadioButton printAll, printRange;
    JLabel, startPage, endPage;
    JTextField onStartField, endField;

    public PrintRangePanel (int start, int end) {
        ButtonGroup group = new ButtonGroup ();

        printAll = new JRadioButton ("Print All");
        printRange = new JRadioButton ("Print Range");

        startPage = new JLabel ("Start Page:");
        endPage = new JLabel ("End Page:");

        startField = new JTextField (Integer.toString (start));
        endField = new JTextField (Integer.toString (end));

        add (printAll); add (printRange);
        add (startPage); add (startField);
        add (endPage); add (endField);

        printRange.setSelected (true);

        group.add (printAll);
        group.add (printRange);

        printRange.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {
                if (printRange.isSelected ()) {
                    startField.setEnabled (true);
                    endField.setEnabled (true);
                    startPage.setEnabled (true);
                    endPage.setEnabled (true);

                    startPage.repaint ();
                    endPage.repaint ();

                    startField.requestFocus ();
                }
            }
        });

        printAll.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent event) {
                if (printAll.isSelected ()) {
                    startField.setEnabled (false);
                    endField.setEnabled (false);
                    startPage.setEnabled (false);
                    endPage.setEnabled (false);

                    startPage.repaint ();
                    endPage.repaint ();
                }
            }
        });
    }
}

```

组件总结 9-3 中对 JRadioButton 类进行了总结。

组件总结 9-3 JRadioButton

模型: JToggleButton.ToggleButtonModel

UI 代表: javax.swing.plaf.basic.BasicRadioButtonUI
绘制器: ——
编辑器: ——
激发的事件: ActionEvent/ChangeEvent/ItemEvent
替代: java.awt.CheckBox
类图: 参见图 8-12

9.4.1 JRadioButton 属性

JRadioButton 类没有自己的属性，它的所有属性都是从 AbstractButton 类继承来的。

9.4.2 JRadioButton 事件

与所有的 Swing 按钮一样，单选钮在激活时激发动作事件，当修改关联属性时激发属性变化事件，并在它们的状态改变时激发变化事件。有关按钮激发的动作事件和变化事件参见 8.4.2 节“JButton 事件”，有关属性变化事件则参见 3.2.4 节。另外，当它们被选取或被取消选取时，单选钮激发子项事件。用一个子项侦听器处理单选钮选取和取消选取的例子参见例 9-7。

9.4.3 JRadioButton 类总结

类总结 9-5 对 JRadioButton 进行了总结。

类总结 9-5 JRadioButton

扩展: JToggleButton

实现: javax.accessibility.Accessible

1. 构造方法

```
public JRadioButton ()  
public JRadioButton (Icon)  
public JRadioButton (Icon, boolean selected)  
public JRadioButton (String)  
public JRadioButton (String, boolean selected)  
public JRadioButton (String, Icon)  
public JRadioButton (String, Icon, boolean selected)
```

与 JToggleButton 和 JCheckBox 类相同，JRadioButton 的构造方法允许用文本和图标的任意组合及选取状态来创建单选钮。图 9-8 中示出的小应用程序与图 9-6 中示出的小应用程序几乎完全相同，不同之处在于使用的是单选钮而不是复选框。

例 9-8 中列出了图 9-8 中示出的小应用程序的代码。

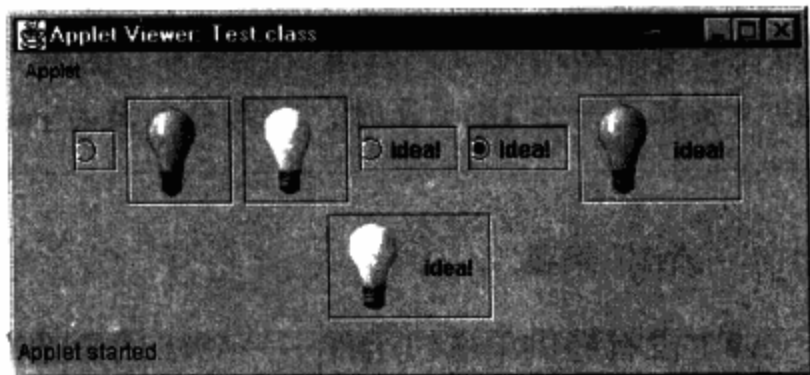


图 9-8 实例化单选钮

例 9-8 实例化单选按钮

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;
```

```

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        Icon icon = new ImageIcon ("bulb.gif");
        JRadioButton [] radioButtons = new JRadioButton [] {
            new JRadioButton (),
            new JRadioButton (icon),
            new JRadioButton (icon, true),
            new JRadioButton ("idea!"),
            new JRadioButton ("idea!", true),
            new JRadioButton ("idea!", icon),
            new JRadioButton ("idea!", icon, true)
        };
        contentPane.setLayout (new FlowLayout ());
        for (int i=0; i < radioButtons.length; ++i) {
            radioButtons [i] .setBorderPainted (true);
            contentPane.add (radioButtons [i]);
            if (radioButtons [i] .getIcon () != null) {
                radioButtons [i] .setSelectedIcon (
                    new ImageIcon ("bulb_bright.gif"));
            }
        }
    }
}

```

与复选框的情况一样，带有一个非 null 图标的单选钮不为选取和取消选取显示控件。因此，应当为带有非 null 图标的单选钮设置一个选定的图标。

2. 方法

```

public String paramString ()
public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()

```

paramString 方法重载 AbstractButton 类中的 paramString 方法，它创建一个字符串，提供与单选钮有关的信息。上面列出的最后三个方法可以在 JComponent 的大多数扩展中找到。Swing 轻量组件能够返回与可访问性有关的内容，包括组件的可访问性信息和它们的 UI 代表的类名。当组件装配了一个 UI 代表时，则调用上面的 updateUI 方法。

9.4.4 AWT 兼容

AWT 没有提供单选钮组件——AWT 的复选框承担了复选框和单选钮的双重任务。AWT 复选框一旦添加到一个复选框组中，就变成了单选钮。

由于 Swing 单选钮与 Swing 复选框除了选取控件外几乎相同，所有复选框与单选钮的 AWT 兼容性是相同的参见复选框一节中“AWT 兼容性”的讨论。

9.5 本章回顾

Swing 的反转按钮、复选框和单选钮都是建立在 AbstractButton 类的基础上的。JToggleButton、JCheckBox 和 JRadioButton 除了含有大量的构造方法和返回信息的方法外，没有其他特别之

处。它们的方法返回与组件的 UI 代表有关的信息。

反转按钮、复选框和单选钮的 Swing 实现比 AWT 的 `CheckBox` 类强大得多。AWT 的 `CheckBox` 类是复选框和单选钮的组合。Swing 按钮既能显示文本，也能显示图像，还能同时显示文本和图像，而 AWT 按钮只能显示文本。Swing 的 `ButtonGroup` 类从实质上说与 AWT 的 `CheckBoxGroup` 类相似，它们均提供相互排斥的选取行为。但是，AWT 的 `CheckBoxGroup` 只能用于 AWT 复选框，而 Swing 的 `ButtonGroup` 适用于 Swing 的所有按钮类型。

第 10 章 菜单和工具条

菜单和工具条是现代用户界面的重要组成部分，Swing 提供了对这两者的完全支持。Swing 提供菜单组件（即菜单条中的菜单和弹出式菜单中的菜单）和菜单项组件（包括复选框和单选按钮菜单项）。

Swing 还提供一个工具条组件，根据工具条的方向，工具条组件包含一行按钮或一列按钮。工具条通常提供对普通特性的简单访问。通常，除键盘捷径键提供对相同功能的访问外，菜单栏和工具条不提供对相同功能集的访问。工具条还可以是悬浮的，这样，可以把工具条在窗口中任意拖动或拖到一个单独的窗口中。

Swing 菜单和菜单项是按钮，因为 JMenuItem 扩展 AbstractButton，JMenu 扩展 JMenuItem，如图 10-1 所示。因此，Swing 菜单和菜单项继承了如下功能，即包含文本和（或）图标、显示光标进入时的图标和助记符等。

图 10-1 示出了 JMenuItem 和 JMenu 的父组件，它们最后都归于 java.awt.Container，以强调菜单和菜单项都是容器。可以把任何类型的组件（从包含一个动画 GIF 的标签到 JTree 的一个实例）添加到一个菜单或菜单项中。

菜单和菜单项还实现 MenuElement 接口，以便参与菜单事件的处理。MenuElement 接口在 10.7 节“菜单元素”中介绍。

工具条和弹出式菜单都扩展 JComponent 类，即任意组件都可以添加到一个工具条或弹出式菜单中。

本章将介绍下面的组件：

- JMenuItem
- JCheckBoxMenuItem
- JRadioButtonMenuItem
- JMenu
- JPopupMenu
- JMenuBar
- JToolBar

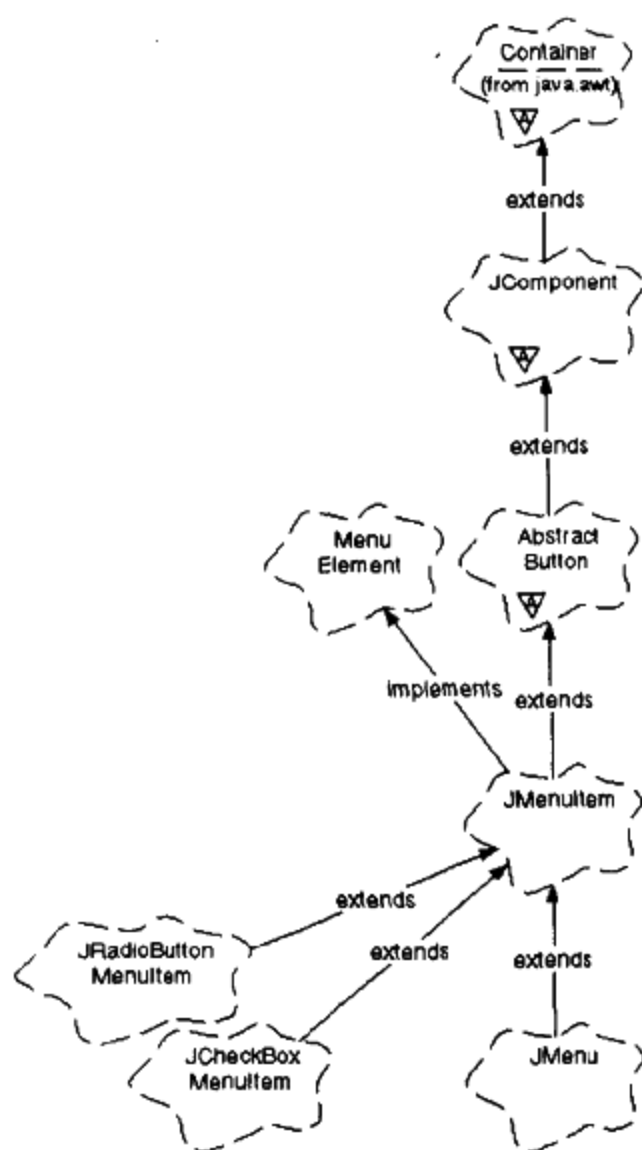


图 10-1 菜单类的层次结构

10.1 菜单、菜单栏和工具条

菜单和工具条有密切的关系，因为通常它们都提供对大多数相同功能的访问。因此，一个或多个动作可以被多个菜单条和工具条所共享。5.3 节“动作”对动作进行了一般性的介绍，5.3.1 节“作为控制中心点的动作”介绍了一个在组件中共享动作的例子。

另外，因为菜单、菜单栏和工具条都是 JComponent 类的扩展，它们都是轻量容器。因此，任何类型的组件都可以添加到一个 Swing 菜单、菜单栏或工具条中。

图 10-2 所示的小应用程序图解说明了在工具条和菜单栏之间共享动作的情况。这个小应用程序还把任意组件以 Swing 单选钮的形式添加到一个工具条、菜单栏和菜单中。

图 10-2 中的左上图显示菜单栏中的一个单选钮，这个按钮正处于光标进入状态，显示的是光标进入时的图像。右上图显示工具条中的另一个单选钮，这个按钮也正处于光标进入状态，显示的是光标进入时的图像。下面的两个图片显示 File 菜单并说明了这样一个事实：一个单选钮可以添加到菜单中，也可以添加到菜单栏和工具条中。

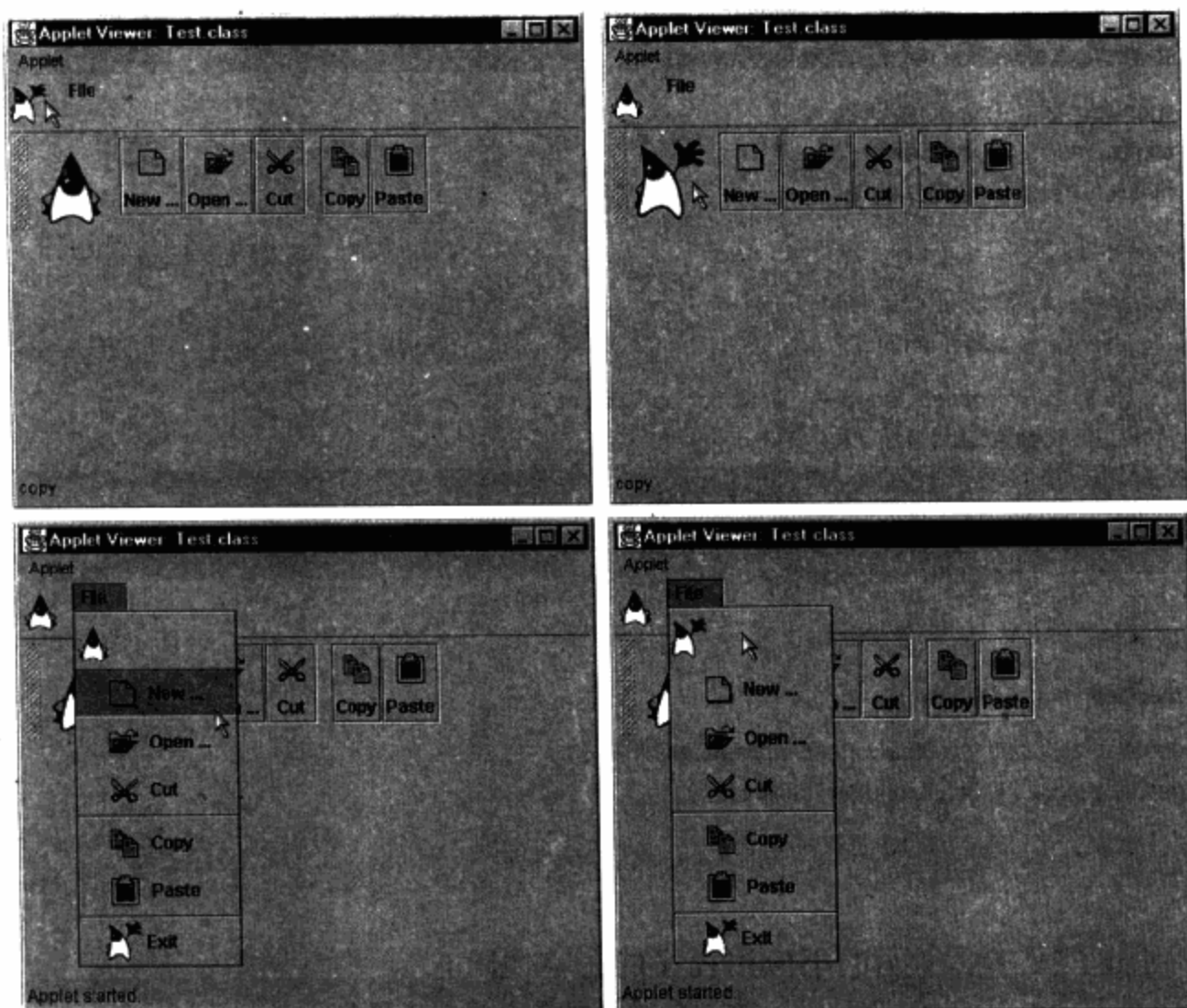


图 10-2 菜单和工具条

这个小应用程序创建了一组被工具条和 File 菜单所共享的动作，并把这些动作添加到菜单和工具条中。退出动作只添加到菜单栏中，因为在工具条中包含一个退出应用程序的按钮是相当成问题的设计思想。在动作添加到工具条和 File 菜单中之前，把单选钮添加到菜单栏、工具条和 File 菜单中。每个单选钮都配备了一个光标进入时的图标。

```
public class Test extends JApplet {
    ...
    public void init () {
        Container contentPane = getContentPane ();
        Action [] actions = {
            new NewAction (),
            new OpenAction (),
            new CutAction (),
            new CopyAction (),
            new PasteAction (),
```

```

        new ExitAction ()
    };
    JToolBar toolbar = new JToolBar ();
    JMenuBar menubar = new JMenuBar ();
    JMenu fileMenu = new JMenu (" File");

    JRadioButton
        menubarDuke = new JRadioButton (dukeStandingSmall),
        menuDuke = new JRadioButton (dukeStandingSmall),
        toolbarDuke = new JRadioButton (dukeStanding);

    menuDuke.setRolloverIcon (dukeWavingSmall);
    menubarDuke.setRolloverIcon (dukeWavingSmall);
    toolbarDuke.setRolloverIcon (dukeWaving);

    menubar.add (menubarDuke);
    toolbar.add (toolbarDuke);
    fileMenu.add (menuDuke);

    ...

    for (int i=0; i < actions.length; ++i) {
        fileMenu.add (actions [i]);

        if (i != actions.length-1) //all but exit action
            toolbar.add (actions [i]);

        if (i == 2 || i == actions.length-2) {
            toolbar.addSeparator ();
            fileMenu.addSeparator ();
        }
    }

    menubar.add (fileMenu);

    contentPane.add (toolbar, BorderLayout.NORTH);
    getRootPane ().setMenuBar (menubar);
}
...
}

```

例 10-1 列出了图 10-2 所示的小应用程序的完整代码。

例 10-1 JRootPane 中的菜单栏和工具条

```

import javax.swing. * ;
import java.awt. * ;
import java.awt.event. * ;

public class Test extends JApplet {
    Icon
        dukeStanding = new ImageIcon ("duke _ standing.gif"),
        dukeWaving = new ImageIcon ("duke _ waving.gif"),
        dukeStandingSmall =
            new ImageIcon ("duke _ standing _ small.gif"),
        dukeWavingSmall = new ImageIcon ("duke _ waving _ small.gif");

    public void init () {
        Container contentPane = getContentPane ();
        Action [] actions = {
            new NewAction (),
            new OpenAction (),

```

```

        new CutAction (),
        new CopyAction (),
        new PasteAction (),
        new ExitAction ()
    };
    JToolBar toolbar = new JToolBar ();
    JMenuBar menubar = new JMenuBar ();
    JMenu fileMenu = new JMenu ("File");

    JRadioButton
        menubarDuke = new JRadioButton (dukeStandingSmall),
        menuDuke = new JRadioButton (dukeStandingSmall),
        toolbarDuke = new JRadioButton (dukeStanding);

    menuDuke.setRolloverIcon (dukeWavingSmall);
    menubarDuke.setRolloverIcon (dukeWavingSmall);
    toolbarDuke.setRolloverIcon (dukeWaving);

    menubar.add (menubarDuke);
    toolbar.add (toolbarDuke);
    fileMenu.add (menuDuke);

    for (int i=0; i < actions.length; ++i) {
        fileMenu.add (actions [i]);

        if (i != actions.length-1)
            toolbar.add (actions [i]);

        if (i == 2 || i == actions.length-2) {
            toolbar.addSeparator ();
            fileMenu.addSeparator ();
        }
    }

    menubar.add (fileMenu);

    contentPane.add (toolbar, BorderLayout.NORTH);
    getRootPane ().setMenuBar (menubar);
}

class NewAction extends AbstractAction {
    public NewAction () {
        super ("New ...", new ImageIcon ("new.gif"));
    }

    public void actionPerformed (ActionEvent event) {
        showStatus ("new");
    }
}

class OpenAction extends AbstractAction {
    public OpenAction () {
        super ("Open ...", new ImageIcon ("open.gif"));
    }

    public void actionPerformed (ActionEvent event) {
        showStatus ("open");
    }
}

class CutAction extends AbstractAction {
    public CutAction () {
        super ("Cut", new ImageIcon ("cut.gif"));
    }
}

```

```
public void actionPerformed (ActionEvent event) {
    showStatus ("out");
}

class CopyAction extends AbstractAction {
    public CopyAction () {
        super ("Copy", new ImageIcon ("copy.gif"));
    }

    public void actionPerformed (ActionEvent event) {
        showStatus ("copy");
    }
}

class PasteAction extends AbstractAction {
    public PasteAction () {
        super ("Paste", new ImageIcon ("paste.gif"));
    }

    public void actionPerformed (ActionEvent event) {
        showStatus ("paste");
    }
}

class ExitAction extends AbstractAction {
    public ExitAction () {
        super ("Exit");
        putValue (Action.SMALL_ICON, dukeWavingSmall);
    }

    public void actionPerformed (ActionEvent event) {
        System.exit (0);
    }
}
```

10.2 菜单和弹出式菜单

当激活 Swing 菜单上的按钮时，则显示一个弹出式菜单。如果一个菜单在菜单栏中，它就称作顶层菜单，而包含在一个菜单中的多个菜单称作右拉式菜单。（右拉式菜单也称作层叠式菜单，有关“右拉式”菜单的更多信息，请参见 10.6.2 节“右拉式菜单”）。

图 10-3 中示出的小应用程序包含一个菜单条，它有两个菜单（一个顶层菜单和一个右拉式菜单）。当“File”菜单被激活时，将显示一个弹出式菜单，这个菜单包含“File”菜单的菜单项。当右拉式菜单项被激活时，将显示另一个弹出式菜单（它包含复选框菜单项）。

例 10-2 列出了图 10-3 所示的小应用程序。

例 10-2 一个简单的菜单举例

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");
```

```

JMenu pullRightMenu = new JMenu ("pull right");

fileMenu.add ("New ...");
fileMenu.add ("Open ...");
fileMenu.add ("Save");
fileMenu.add ("Save As ...");
fileMenu.addSeparator ();
fileMenu.add (pullRightMenu);
fileMenu.add ("Exit");

pullRightMenu.add (new JCheckBoxMenuItem ("Bush"));
pullRightMenu.add (new JCheckBoxMenuItem ("Tonic"));
pullRightMenu.add (new JCheckBoxMenuItem ("Radio Head"));
pullRightMenu.add (new JCheckBoxMenuItem ("Marcy Playground"));
pullRightMenu.add (new JRadioButtonMenuItem ("Silver Chair"));

mb.add (fileMenu);
setJMenuBar (mb);

```

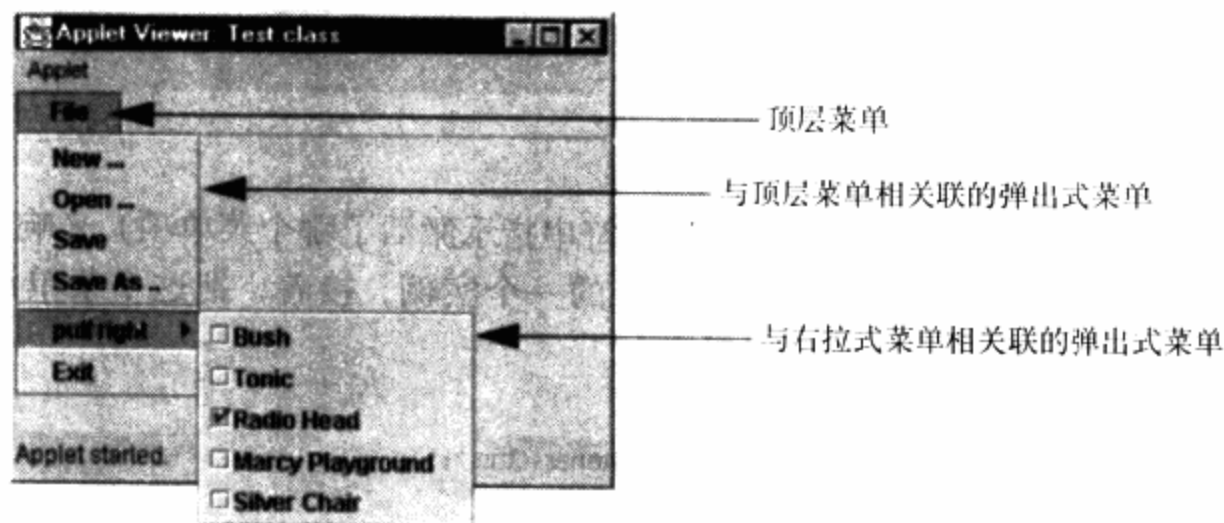


图 10-3 一个 File 菜单的例子

这个小应用程序创建了一个 `JMenuBar` 实例和两个 `JMenu` 实例。通过调用 `JMenu.add (String)` 把菜单项添加到菜单中。然后，把 `File` 菜单添加到菜单栏中，再通过调用 `JApplet.setJMenuBar (JMenuBar)` 把这个菜单栏添加到这个小应用程序中。

注意 图 10-3 所示的小应用程序不处理菜单事件；例如，激活“Exit”菜单项将不退出这个小应用程序。在本章的其他部分将介绍菜单事件的处理。（要退出图 10-3 所示的小应用程序，单击这个小应用程序窗口中的关闭按钮）。

10.3 JMenuItem

菜单项是按钮，因为 `JMenuItem` 扩展 `AbstractButton`，因此，菜单项可以显示文本和图标。菜单项还继承了在它们激活时激发动作事件的能力。因为 `AbstractButton` 最终扩展 `java.awt.Container`，所以菜单项还可以包含任意类型的 AWT 组件和 Swing 组件。

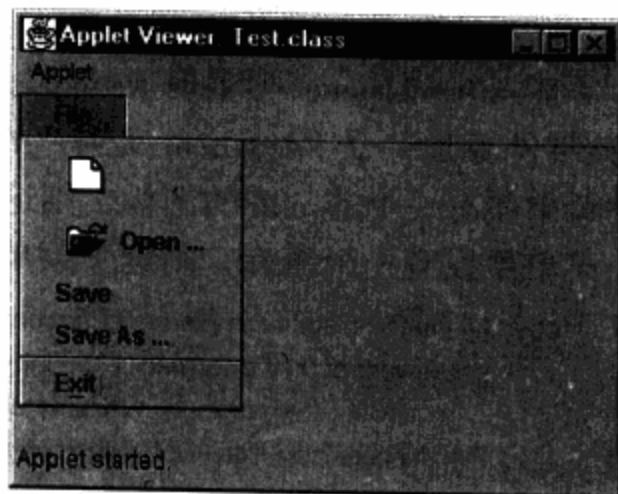


图 10-4 创建 JMenuItem 实例

图 10-4 显示了一个创建五个菜单项的小应用程序。

这个小应用程序实例化两个图标、一个菜单栏和一个具有五个菜单项的 File 菜单。在创建了这些菜单项后，这些菜单项添加到 File 菜单中。

```
public class Test extends JApplet {
    public void init () {
        Icon newIcon = new ImageIcon ("new.gif",
                                      "Create a new document");
        Icon openIcon = new ImageIcon ("open.gif",
                                       "Open an existing document");

        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");

        JMenuItem newItem = new JMenuItem (newIcon);
        JMenuItem openItem = new JMenuItem ("Open ...", openIcon);
        JMenuItem saveItem = new JMenuItem ("Save");
        JMenuItem saveAsItem = new JMenuItem ("Save As ...");
        JMenuItem exitItem = new JMenuItem ("Exit", 'x');

        fileMenu.add (newItem);
        fileMenu.add (openItem);
        fileMenu.add (saveItem);
        fileMenu.add (saveAsItem);
        fileMenu.addSeparator ();
        fileMenu.add (exitItem);
        ...
    }
}
```

这个小应用程序通过显示一个字符串（该字符串指示激活了哪个菜单项）来响应激活菜单项的动作。每个菜单项都配备了 `MenuItemListener` 的一个实例，接着，把这个菜单添加到菜单栏中，再把这个菜单栏添加到这个小应用程序中。

```
...
MenuItemListener listener = new MenuItemListener (this);
newItem.addActionListener (listener);
openItem.addActionListener (listener);
saveItem.addActionListener (listener);
saveAsItem.addActionListener (listener);
...
mb.add (fileMenu);
setJMenuBar (mb);
}
```

`MenuItemListener` 类实现 `ActionListener` 接口，通过实现 `actionPerformed` 方法来获得对事件源（该事件源是一个 `JMenuItem` 实例）的一个引用，和对菜单项的图标的一个引用。如果被激活的菜单项有一个非 `null` 的图标，则这个图标的描述显示在这个小应用程序的状态区中。如果这个菜单项没有一个图标，则这个菜单项的文本显示在这个小应用程序的状态区中。

```
class MenuItemListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JMenuItem item = (JMenuItem) e.getSource ();
        ImageIcon icon = (ImageIcon) item.getIcon ();
        if (icon != null)
            System.out.println (icon.getDescription ());
    }
}
```



```

else
    System.out.println (item.getText ());

```

例 10-3 列出了图 10-4 示出的小应用程序的代码。

例 10-3 安装一些菜单项

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Icon newIcon = new ImageIcon ("new.gif",
                                       "Create a new document");
        Icon openIcon = new ImageIcon ("open.gif",
                                       "Open a existing document");

        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");

        JMenuItem newItem = new JMenuItem (newIcon);
        JMenuItem openItem = new JMenuItem ("Open ...", openIcon);
        JMenuItem saveItem = new JMenuItem ("Save");
        JMenuItem saveAsItem = new JMenuItem ("Save As ...");
        JMenuItem exitItem = new JMenuItem ("Exit", 'x');

        fileMenu.add (newItem);
        fileMenu.add (openItem);
        fileMenu.add (saveItem);
        fileMenu.add (saveAsItem);
        fileMenu.addSeparator ();
        fileMenu.add (exitItem);

        MenuItemListener listener = new MenuItemListener (this);
        newItem.addActionListener (listener);
        openItem.addActionListener (listener);
        saveItem.addActionListener (listener);
        saveAsItem.addActionListener (listener);
        exitItem.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                System.exit (0);
            }
        });

        mb.add (fileMenu);
        setJMenuBar (mb);
    }
}

class MenuItemListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JMenuItem item = (JMenuItem) e.getSource ();
        ImageIcon icon = (ImageIcon) item.getIcon ();
        if (icon != null)
            System.out.println (icon.getDescription ());
    }
}

```

```

else
    System.out.println (item.getText ());
}

```

10.3.1 菜单项快捷键和助记符键

可以为菜单项指定一个快捷键，这个快捷键由 `KeyStroke` 的一个实例表示。快捷键与助记符键类似，因为它们都代表键盘捷径方法，用来激活与一个菜单项相关联的一个动作。然而，助记符键要与由组件的界面样式定义的 `Meta` 键一起按下。例如，图 10-4 所示的小应用程序为 `Exit` 菜单项指定了一个“x”助记符键。在 Windows 界面样式中，助记符键必须与 `Alt` 键一起按下来激活这个菜单项。相反，快捷键由 `KeyStroke` 的一个实例指定，这个实例指定准确的键组合，按下这个组合键才能激活菜单项。

为了进一步说明菜单项快捷键和助记等键的使用，图 10-5 所示的应用程序^① 为 `File` 菜单和 `Exit` 菜单项指定了助记符键，为 `Exit` 菜单项指定了一个快捷键。

用户可以通过按下 `Meta + F` 来退出这个应用程序，`Meta + F` 激活 `File` 菜单，然后按下 `Meta + X`，它激活 `Exit` 菜单项。我们知道，`Meta` 键与界面样式有关，例如，`Metal` 界面样式的 `Meta` 键是 `Alt` 键。通过按下 `Meta + F` / `Meta + X` 来退出这个应用程序使用的是助记符键的方法，它经过了两个菜单。

还可以利用与 `Exit` 菜单项相关联的快捷键来退出这个应用程序。即按下 `Alt + X`，就可退出这个应用程序。注意，使用快捷键不经过数个菜单，快捷键直接调用与菜单项相关联的动作。

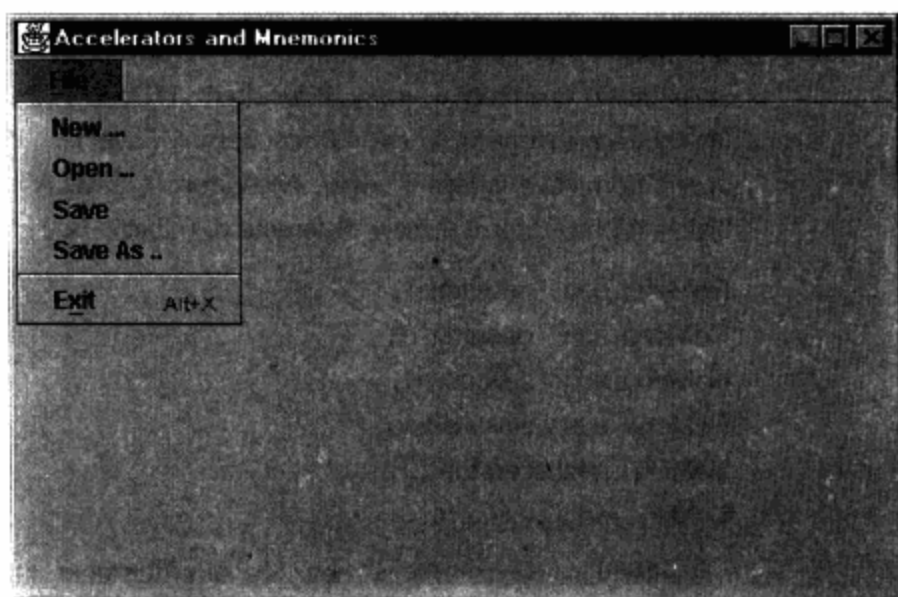


图 10-5 菜单项的助记符键和快捷键

快捷键是这样指定的，首先从 `static KeyStroke.getKeyStroke` 方法中获得对一个 `KeyStroke` 实例的引用，然后，把这个键击传送给 `JMenuItem.setAccelerator` 方法。这个键击指定“x”键必须与 `Alt` 键一起按下。

通过使用 `AbstractButton.setMnemonics` 方法来指定助记符键。为说明起见，我们调用以一个字符为参数的 `AbstractButton.setMnemonic (char)` 来指定 `File` 菜单的助记符键，而用 `AbstractButton.setMnemonic (int)` 来指定 `Exit` 菜单项的助记符键。通常，用 `KeyEvent` 类中的 `integer` 值来指定助记符键比用一个字符来指定助记符键要好，因为使用 `KeyEvent` 类中的一个常量确保了助记符键在国际化的小应用程序和应用程序中都能工作正常。

```

public class Test extends JFrame {
    public Test () {
        Container contentPane = getContentPane ();

        JMenuBar mmb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");

```

① 由于 AWT 的一个焦点错误，图 10-5 所示的应用程序将不能像一个小应用程序那样正常工作。

```

JMenuItem exitItem = new JMenuItem ("Exit");

fileMenu.add ("New ...");
fileMenu.add ("Open ...");
fileMenu.add ("Save");
fileMenu.add ("Save As ...");
fileMenu.addSeparator ();
fileMenu.add (exitItem);

exitItem.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        System.exit (0);
    }
});

KeyStroke ks = KeyStroke.getKeyStroke (KeyEvent.VK_X,
                                         Event.ALT_MASK);

exitItem.setAccelerator (ks);

fileMenu.setMnemonic ('F');
exitItem.setMnemonic (KeyEvent.VK_X);

mb.add (fileMenu);
setJMenuBar (mb);
}

```

例 10-4 列出了图 10-5 所示的应用程序的代码。

例 10-4 带助记符键和快捷键的菜单项

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    public Test () {
        Container contentPane = getContentPane ();

        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");
        JMenuItem exitItem = new JMenuItem ("Exit");

        fileMenu.add ("New ...");
        fileMenu.add ("Open ...");
        fileMenu.add ("Save");
        fileMenu.add ("Save As ...");
        fileMenu.addSeparator ();
        fileMenu.add (exitItem);

        exitItem.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                System.exit (0);
            }
        });

        KeyStroke ks = KeyStroke.getKeyStroke (KeyEvent.VK_X,
                                                Event.ALT_MASK);

        exitItem.setAccelerator (ks);

        fileMenu.setMnemonic ('F');
        exitItem.setMnemonic (KeyEvent.VK_X);

        mb.add (fileMenu);
    }
}

```

```
setJMenuBar (mb);  
  
public static void main (String args []) {  
    GJApp.launch (new Test (),  
        "Accelerators and Mnemonics", 300, 300, 450, 300);  
}
```

Swing 菜单项由 JMenuItem 类表示，组件总结 10-1 总结了这个类。

组件总结 10-1 JMenuItem

- 模型： ButtonModel
- UI 代表： swing.plaf.basic.BasicMenuItemUI
- 绘制器： ——
- 编辑器： ——
- 激发的事件： ActionEvent/ChangeEvent/PropertyChangeEvent/MenuDragMouseEvent
- 替换： java.awt.MenuItem
- 类图：

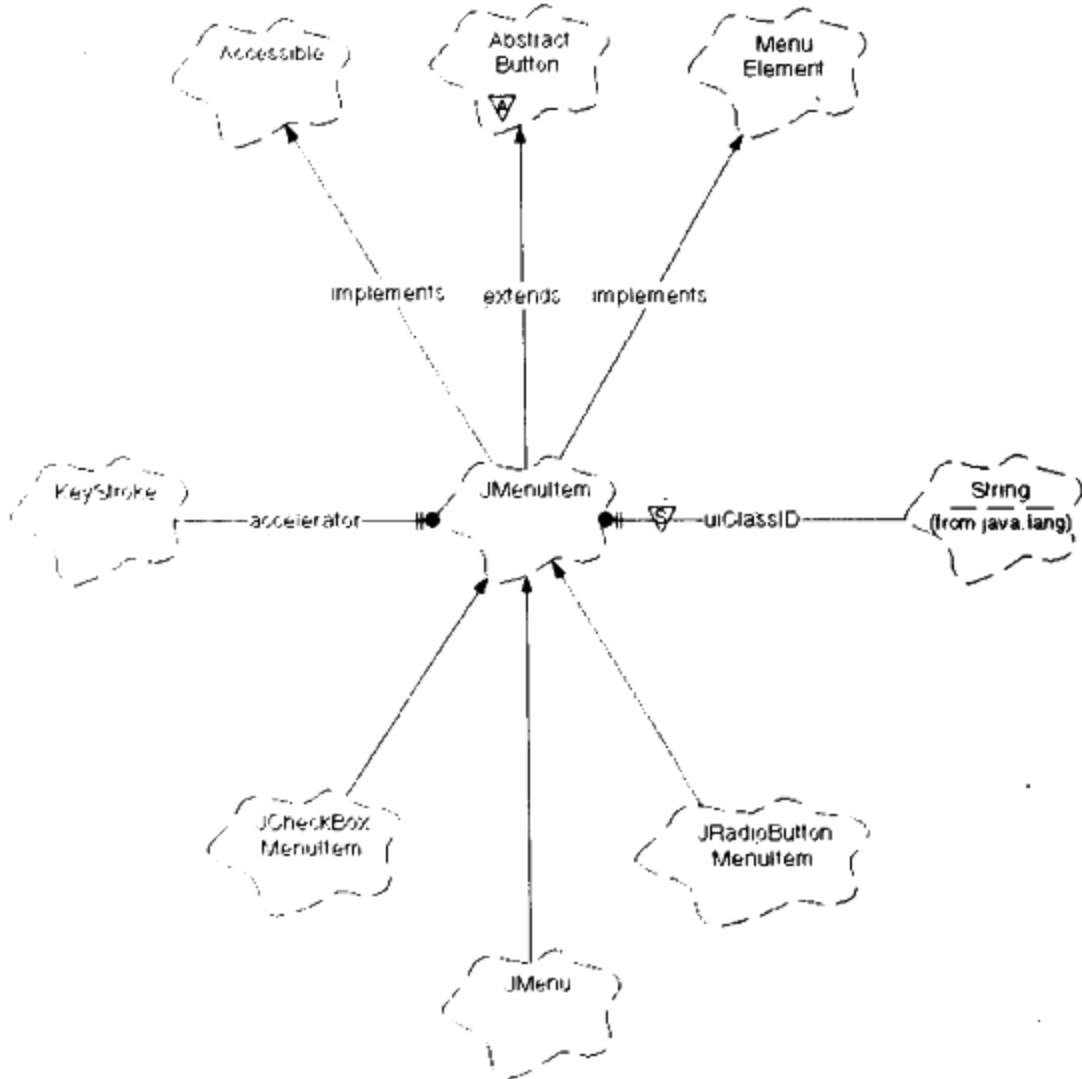


图 10-6 JMenuItem 类图

JMenuItem 扩展 AbstractButton，并实现 Accessible 接口和 MenuElement 接口。JMenuItem 是 JCheckBoxMenuItem、JRadioButtonMenuItem 和 JMenu 的超类。JMenu 扩展 JMenuItem 的事实允许菜单被指定为菜单项，因此，产生了右拉式菜单。

因为 JMenuItem 扩展 AbstractButton，所以菜单项（和菜单）完全可以以按钮的形式进行操

纵。有关操纵 `AbstractButton` 的实例的方式的更多信息，请参见 8.4 节“`JButton`”。每个 `JMenuItem` 的实例可以配备一个快捷键。快捷键是用于激活一个菜单项的键盘捷径键。快捷键由 `KeyStroke` 的一个实例来代表，这个实例允许开发人员指定要激活一个菜单项的组合键。

Swing 提示

Swing 的菜单和菜单项都是按钮

记住 Swing 的菜单和菜单项最终都要扩展 `AbstractButton` 类是很重要的。这就是说，`AbstractButton` 类（参见 8.4.2 节“`JButton`”）的所有属性、方法和事件也能应用于菜单和菜单项中。例如，缺省时，Swing 菜单的水平文本位置设置为 `SwingConstants.LEFT`，从图 10-4 中可以看出。因为 `JMenu` 类扩展 `AbstractButton`，所以，可以用 `AbstractButton.setVerticalTextPosition()` 来改变一个菜单的缺省水平文本位置。

10.3.2 JMenuItem 属性

除继承的属性外，`JMenuItem` 类自己只有一个属性——快捷键属性，这个属性在例 10-4 中使用了。有关 `JMenuItem` 从 `AbstractButton` 继承的属性列表，请参见表 8-3。

10.3.3 JMenuItem 事件

与所有的 Swing 按钮一样，激活菜单项时将激发动作事件，当修改它们的关联属性时将激发属性变化事件，在它们的状态改变时将激发变化事件。有关被按钮激发的动作事件和变化事件的更多信息，请参见 8.4.2 节“`JButton` 事件”。有关属性变化事件的更多信息，请参见 3.2.4 节。另外，当菜单项选取或取消选取时，这些菜单项会激发菜单项事件。有关处理子项事件的更多信息，请参见 9.1.2 节“`JToggleButton` 事件”。

当鼠标拖动到一个菜单项上时，菜单项还激发 `MenuDragMouseEvents`，在一个菜单项处于待命状态时，按下、释放、输入一个键都将使菜单项激发 `MenuKeyEvents` 事件。

1. 菜单变化事件

图 10-7 所示的小应用程序监听由 `Exit` 菜单项激发的动作事件并通过退出这个小应用程序来进行响应。这个小应用程序还监听每个菜单项的变化事件，而且在一个菜单项处于待命状态时显示与此菜单项相关联的动作命令。一个菜单项处于待命状态

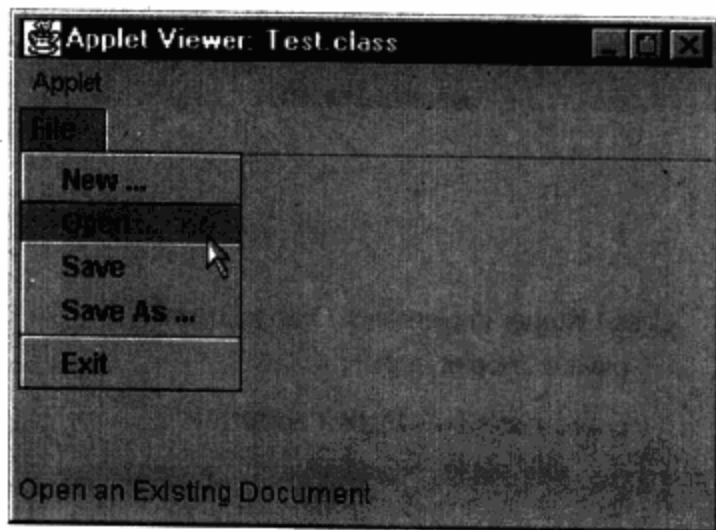


图 10-7 菜单项变化事件

是指这个菜单项已选取但没有激活，例如，图 10-7 所示的 `Open...` 菜单项就处于待命状态。

例 10-5 列出了图 10-7 示出的小应用程序的代码。

例 10-5 监听菜单项动作和菜单项变化事件

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

```

public class Test extends JApplet {
    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");
        JMenuItem newItem = new JMenuItem ("New ..."),
            openItem = new JMenuItem ("Open ..."),
            saveItem = new JMenuItem ("Save"),
            saveAsItem = new JMenuItem ("Save As ..."),
            exitItem = new JMenuItem ("Exit");

        Listener listener = new Listener (this);

        fileMenu.add (newItem);
        fileMenu.add (openItem);
        fileMenu.add (saveItem);
        fileMenu.add (saveAsItem);
        fileMenu.addSeparator ();
        fileMenu.add (exitItem);

        newItem.setActionCommand ("Create a New Document");
        openItem.setActionCommand ("Open an Existing Document");
        saveItem.setActionCommand ("Save Document");
        saveAsItem.setActionCommand ("Save Document As ...");
        exitItem.setActionCommand ("Exit the applet");

        newItem.addChangeListener (listener);
        openItem.addChangeListener (listener);
        saveItem.addChangeListener (listener);
        saveAsItem.addChangeListener (listener);
        exitItem.addChangeListener (listener);

        mb.add (fileMenu);
        setJMenuBar (mb);

        exitItem.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                System.exit (0);
            }
        });
    }
}

class Listener implements ChangeListener {
    private JApplet applet;

    public Listener (JApplet applet) {
        this.applet = applet;
    }

    public void stateChanged (ChangeEvent e) {
        JMenuItem b = (JMenuItem) e.getSource ();

        if (b.isArmed ())
            applet.showStatus (b.getActionCommand ());
    }
}

```

这个小应用程序创建一个菜单栏、File 菜单和它的菜单项。接着把这些菜单项添加到 File 菜单中，并设置每个菜单项的动作命令。这个小应用程序然后创建 Listener 的一个实例（一个实现 ChangeListener 接口的事件监听器）并把这个监听器添加到每个菜单项中。当 exitItem 激发

一个 `ActionEvent` 时, `exitItem` 用添加到 `exitItem` 的动作监听器来终止这个小应用程序。

`Listener` 类监听菜单项变化事件并通过获得对激发这个事件的菜单项的一个引用来进行响应。如果菜单项处于待命状态, 则获取它的动作命令并把这个动作命令显示在这个小应用程序的状态区中。

2. 菜单的拖动鼠标事件

当在一个菜单元素上拖动鼠标光标时, 则把 `MenuDragMouseEvent` 的实例发送给已向菜单项登记了的、处于监听状态中的 `MenuDragMouseListener`。接口总结 10-1 总结了 `MenuDragMouseListener` 接口。

接口总结 10-1 `MenuDragMouseListener`

```
public abstract void menuDragMouseDragged (MenuDragMouseEvent)
public abstract void menuDragMouseEntered (MenuDragMouseEvent)
public abstract void menuDragMouseExited (MenuDragMouseEvent)
public abstract void menuDragMouseReleased (MenuDragMouseEvent)
```

上面所列的第一组方法分别为三种动作激活: 鼠标光标在一个菜单元素上拖动、鼠标光标进入一个菜单元素和鼠标光标从一个菜单元素中出来。当在一个菜单元素上释放鼠标按钮时, 将激活最后一行方法。

上面所列的所有方法都以 `MenuDragMouseEvent` 的一个实例为参数, 类总结 10-1 总结了 `MenuDragMouseEvent`。

类总结 10-1 `MenuDragMouseEvent`

扩展: `java.awt.event.MouseEvent`

1. 构造方法

```
public MenuDragMouseEvent (Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean
                             popupTrigger, MenuElement [] path, MenuSelectionManager mgr)
```

`MenuDragMouseEvent` 类只提供了一个构造方法, 它以大量的事件信息为参数, 这些信息包括修饰键和与这个事件有关的点击次数。开发人员很少 (如果有的话) 实例化 `MenuDragMouseEvent` 的实例。

2. 方法

```
public MenuSelectionManager getMenuSelectionManager ()
public MenuElement [] getPath ()
```

`MenuDragMouseEvent` 类提供上面所列的两种方法, 它们返回对菜单选取管理器的引用和一个菜单元素数组, 这个数组指定到达发生事件的菜单元素的路径。

菜单选取管理器是菜单选取的实现细节, 实际中很少使用 `getMenuSelectionManager` 方法。从 `getPath` 方法中返回的数组中的最后菜单元素代表发生事件的菜单元素。

可以用一个菜单的拖动鼠标监听器而不是一个变化监听器来重写图 10-5 所列的小应用程序。除 `Listener` 类外, 这个小应用程序的其他部分可以不变化, 可以像下面这样来实现 `Listener` 类:

```
class Listener implements MenuDragMouseListener {
    private JApplet applet;
    public Listener (JApplet applet) {
        this.applet = applet;
    }
    public void menuDragMouseEntered (MenuDragMouseEvent e) {
```



```

|
public void menuDragMouseDragged (MenuDragMouseEvent e) {
    MenuElement [] path = e.getPath ();
    MenuElement lastPathElement = path [path.length-1];
    JMenuItem menuItem =
        (JMenuItem) lastPathElement.getComponent ();

    applet.showStatus (menuItem.getActionCommand ());
}
public void menuDragMouseExited (MenuDragMouseEvent e) {
}
public void menuDragMouseReleased (MenuDragMouseEvent e) {
}
|

```

menuDragMouseDragged 方法获得到达发生事件的菜单元素的路径，它首先调用 MenuDragMouseEvent.getPath 方法，然后进行一些操作以获得对菜单项的一个引用。在例 10-5 所列的小应用程序中，菜单项的动作命令显示在这个小应用程序的状态区中。

注意 在 Swing 1.1 FCS 中，menuDragMouseEntered、menuDragMouseExited 和 menuDragMouseReleased 方法从来没有调用过。

10.3.4 JMenuItem 类总结

类总结 10-2 列出了 JMenuItem 的 public 和 protected 变量和方法。

类总结 10-2 JMenuItem

扩展：AbstractButton

实现：javax.accessibility.Accessible、MenuElement

1. 构造方法

```

public JMenuItem ()
public JMenuItem (Icon)
public JMenuItem (String)
public JMenuItem (String, Icon)
public JMenuItem (String, int mnemonic)

```

JMenuItem 提供五个构造方法来实例化具有不同配置（即配置不同的字符串和图标）的菜单项。上面所列的最后一个构造方法以 integer 值为参数，该值指定这个菜单项的助记符键。

注意 以字符串和图标为参数构造的菜单项不能在构造时指定一个助记符键，然而，可以在构造后通过调用 AbstractButton.setMnemonic (int) 或 AbstractButton.setMnemonic (char) 来指定助记符键。

无参数构造方法创建没有字符串或图标的一个菜单项。无参数构造方法主要在 JavaBeans 编译器工具中使用。

2. 方法

(1) 初始化

```
protected void init (String, Icon)
```

与大多数其他 Swing 组件一样，JMenuItem 实现初始化组件的一个 protected init 方法。JMenuItem.init () 为菜单项设置布局管理器，为菜单项设置文本和图标，并把焦点监听器添加到这个菜单项中。如果需要，JMenuItem 的扩展可以重载 init 方法，但是实际中很少重载这个方法。

(2) MenuElement 接口

```

Public Component getComponent ()
Public MenuElement [] getSubElements ()
Public void menuSelectionChanged (boolean)
Public void processKeyEvent (KeyEvent, MenuElement [], MenuSelectionManager)
Public void processMouseEvent (MouseEvent, MenuElement [], MenuSelectionManager)

```

JMenuItem 实现 MenuElement 接口，这个接口由上面所列的五个方法定义。JMenuItem.processMouseEvent 方法把 MenuDragMouseEvent 发送给已登记的 MenuDragMouseListeners。有关菜单的拖动鼠标，请参见 10.3.3 节。JMenuItem.menuSelectionChanged () 是另一个有趣的 MenuElement 方法，它由 JMenuItem 来实现。如果选取菜单项，则 JMenuItem.menuSelectionChanged () 使这个菜单项处于待命状态。要了解如何实现 MenuElement 接口，请参见 10.7 节。

(3) 快捷键/待命/启用

```

public KeyStroke getAccelerator ()
public void setAccelerator (KeyStroke)

public void setArmed (boolean)
public boolean isArmed ()

public void setEnabled (boolean)

```

可以用上面所列的这些方法来设置和获取一个菜单项的快捷键，并设置一个菜单的允许状态。还可以用 setArmed 方法用程序方式使菜单项处于待命状态，可以用 isArmed 方法来确定一个菜单是否处于待命状态。

(4) 监听器/事件激发

```

public void addMenuDragMouseListener (MenuDragMouseListener)
public void addMenuKeyListener (MenuKeyListener)

public void removeMenuDragMouseListener (MenuDragMouseListener)
public void removeMenuKeyListener (MenuKeyListener)

public void processMenuDragMouseEvent (MenuDragMouseEvent)

protected void fireMenuDragMouseDragged (MenuDragMouseEvent)
protected void fireMenuDragMouseEntered (MenuDragMouseEvent)
protected void fireMenuDragMouseExited (MenuDragMouseEvent)
protected void fireMenuDragMouseReleased (MenuDragMouseEvent)
protected void fireMenuKeyPressed (MenuKeyEvent)
protected void fireMenuKeyReleased (MenuKeyEvent)
protected void fireMenuKeyTyped (MenuKeyEvent)

```

上面所列的这些方法中，除了那些把事件发送给菜单的拖动鼠标监听器和菜单的键监听器的方法外，其他的都是添加和删除菜单的拖动鼠标监听器和菜单的键监听器的方法。

(5) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()
public void setUI (MenuItemUI)

```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

Swing 提示

快捷键与助记符键的区别

快捷键和助记符键都代表键盘捷径方法，因此，有必要指出它们之间的差别。

助记符键要与特定界面样式的“Meta”键一起使用。助记符键只有一个字符，如“x”，但是“x”必须与“Meta”键一起按下。例如，在 Windows 界面样式中，“Meta”键是 Alt 键，因此，指定为“x”的助记符键必须与 Alt 键一起按下，才能激活与之相关联的组件。

快捷键也是一种键盘捷径键，然而，特定界面样式的“Meta”键与快捷键无关。助记符键是一个字符或一个整数值，而快捷键是由 KeyStroke 的一个实例指定的，这个实例指定用来激活与之相关联的组件的完整的键组合。最后，助记符键用于激活菜单，而快捷键不激活菜单；快捷键只调用与一个菜单项相关联的动作而不显示菜单。

10.3.5 AWT 兼容

表 10-1 列出了由 java.awt.Menuitem 实现的 public 方法及由 JMenuItem 实现的相应方法。

表 10-1 java.awt.Menuitem 的方法和 JMenuItem 的对应方法^①

java.awt.Menuitem 的方法	JMenuItem 的对应方法
void addActionListener (ActionListener)	void addActionListener (ActionListener)
void disableEvents (long)	void disableEvents (long)
void deleteShortcut ()	void setAccelerator (null)
void enableEvents (long)	void enableEvents (long)
String getActionCommand ()	String getActionCommand ()
String getLabel ()	String getLabel ()
MenuShortcut getShortcut ()	KeyStroke getAccelerator ()
boolean isEnabled ()	boolean isEnabled ()
String paramString ()	String paramString ()
void removeActionListener ()	void removeActionListener ()
void setActionCommand (String)	void setActionCommand (String)
void setEnabled (boolean)	void setEnabled (boolean)
void setLabel (String)	void setLabel (String)
void setShortcut (MenuShortcut)	void setAccelerator (KeyStroke)

① 对具有不同原型的方法用黑体字加重表示。

Swing 菜单项与 AWT 菜单项的源代码几乎完全兼容。两者的主要差别是 AWT 菜单项用 java.awt.MenuShortcut 的一个实例来指定快捷键，而 Swing 菜单项用 KeyStroke 的一个实例来指定快捷键。当把 AWT 菜单项的代码转换为 Swing 的菜单项的代码时，指定快捷键的方法如下所示：

```
// AWT Menu Items;
menuItem.setShortcut (new MenuShortcut ('x'));

//true means that SHIFT-x activates the menu item
menuItem.setShortcut (new MenuShotcut ('x', true));

//Swing Equivalent;
menuItem.setAccelerator (KeyStroke.getKeyStroke (KeyEvent.VK_X));
menuItem.setAccelerator (KeyStroke.getKeyStroke (KeyEvent.VK_X, Event.SHIFT_MASK));
```

上面的第二种捷径键是用带两个参数的 MenuShortcut 构造方法来创建的，这个方法的一个参数是激活这个菜单项的键，第二个参数（boolean 值）指定是否必须使用 Shift 修饰键来激活这个菜单项。与之对应的 Swing 方法获得对 KeyStroke 的一个实例的引用，这个实例指定 Event.SHIFT_MASK 作为必须与“x”键一起按下的修饰键。

10.4 JCheckBoxMenuItem

JCheckBoxMenuItem 是 JMenuItem 的一个简单扩展，它绘制一个复选框控制，缺省时，这个复选框在菜单项文本的左边[⊖]。

图 10-8 示出了一个小应用程序，它创建了三个复选框菜单项。

“eagle”复选框菜单项在构造时刻被选取，“ladybug”复选框菜单项在构造后设置了一个助记键。

例 10-6 列出了图 10-8 示出的小应用程序的代码。

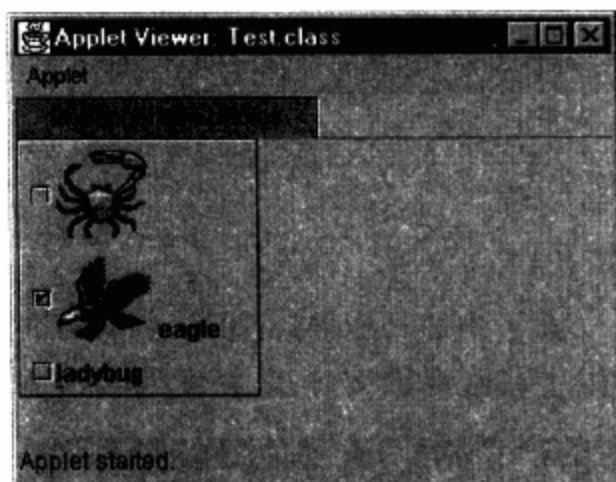


图 10-8 创建 JCheckBoxMenuItem 的实例

例 10-6 实例化复选框菜单项

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();

        JMenuBar mb = new JMenuBar ();
        JMenu checkBoxMenu = new JMenu ("Endangered Species");

        ImageIcon crabIcon = new ImageIcon ("crab.gif");
        ImageIcon eagleIcon = new ImageIcon ("eagle.gif");

        JMenuItem
            crabItem = new JCheckBoxMenuItem (crabIcon),
            eagleItem = new JCheckBoxMenuItem ("eagle",
                                                eagleIcon, true),
            ladybugItem = new JCheckBoxMenuItem ("ladybug");

        checkBoxMenu.add (crabItem);
        checkBoxMenu.add (eagleItem);
        checkBoxMenu.add (ladybugItem);

        ladybugItem.setMnemonic ('l');

        mb.add (checkBoxMenu);
        setJMenuBar (mb);
    }
}
```

组件总结 10-2 总结了 JCheckBoxMenuItem 类。

组件总结 10-2 JCheckBoxMenuItem

模型:	JToggleButton.ToggleButtonModel
UI 代表:	swing.plaf.basic.BasicCheckBoxMenuItemUI
绘制器:	——
编辑器:	——

⊖ 所有的 Swing 界面样式都把控制画在左边。

激发的事件: `ActionEvent/ChangeEvent/ItemEvent`
替换: `java.awt.CheckboxMenuItem`
类图: 参见图 10-6

10.4.1 JCheckBoxMenuItem 属性

源于 `JCheckBoxMenuItem` 的唯一的属性是 `state` 属性, 它指示是否选取了一个复选框菜单项。提供了获取和设置 `state` 属性的一些方法。例 10-7 说明了如何确定 `JCheckBoxMenuItem` 的选取状态。

10.4.2 JCheckBoxMenuItem 事件

与所有的 Swing 按钮一样, 激活复选框菜单项时将激发动作事件, 当修改它们的关联属性时将激发属性改变事件, 当它们的状态改变时, 将激发变化事件。有关按钮激发的动作事件和变化事件的更多信息, 请参见 8.4.2 节“`JButton` 事件”, 有关属性变化的更多信息, 请参见 3.2.4 节。另外, 当选取或取消选取复选框时, 复选框菜单项将激发菜单项事件。有关处理菜单项事件的更多信息, 请参见 9.1.2 节“`JToggleButton` 事件”。

图 10-9 所示的小应用程序说明对复选框菜单项状态的访问。这个小应用程序实例化 `JCheckBoxMenuItem` 的三个实例, 并把一个动作监听器添加到每个菜单项中。当其中的一个菜单项激活时, 这个监听器就在这个小应用程序的状态区中显示每个菜单项的状态。

例 10-7 列出了图 10-9 示出的小应用程序的代码。

例 10-7 访问 `JCheckBoxMenuItem` 状态

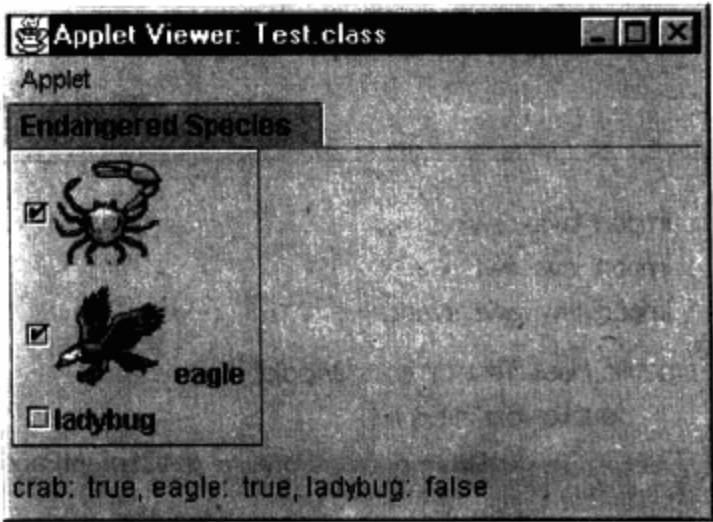


图 10-9 JCheckBoxMenuItem 状态

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private ImageIcon crabIcon = new ImageIcon ("crab.gif");
    private ImageIcon eagleIcon = new ImageIcon ("eagle.gif");

    private JCheckBoxMenuItem
        crabItem = new JCheckBoxMenuItem (crabIcon),
        eagleItem = new JCheckBoxMenuItem ("eagle", eagleIcon),
        ladybugItem = new JCheckBoxMenuItem ("ladybug");

    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu checkBoxMenu = new JMenu ("Endangered Species");
        Listener listener = new Listener ();

        checkBoxMenu.add (crabItem);
        checkBoxMenu.add (eagleItem);
        checkBoxMenu.add (ladybugItem);
```

```

        crabItem.addActionListener (listener);
        eagleItem.addActionListener (listener);
        ladybugItem.addActionListener (listener);

        mb.add (checkBoxMenu);
        setJMenuBar (mb);
    }

    class Listener implements ActionListener {
        public void actionPerformed (ActionEvent e) {
            showStatus ("crab: " + crabItem.getState () + ", " +
                "eagle: " + eagleItem.getState () + ", " +
                "ladybug: " + ladybugItem.getState ());
        }
    }

```

10.4.3 JCheckBoxMenuItem 类总结

类总结 10-3 列出了 JCheckBoxMenuItem 的 public 和 protected 变量和方法。

类总结 10-3 JCheckBoxMenuItem

扩展: JMenuItem

实现: javax.accessibility.Accessible、SwingConstants

1. 构造方法

```

public JCheckBoxMenuItem ()
public JCheckBoxMenuItem (Icon)
public JCheckBoxMenuItem (String)
public JCheckBoxMenuItem (String, boolean selected)
public JCheckBoxMenuItem (String, Icon)
public JCheckBoxMenuItem (String, Icon, boolean selected)

```

JCheckBoxMenuItem 的构造方法与 JMenuItem 的构造方法相似, 但 JCheckBoxMenuItem 的实例可以构造为是选取的或是取消选取的 (因此, 把 boolean 参数传送给上面所列的第四个和第六个构造方法)。

JCheckBoxMenuItem 与 JMenuItem 构造方法之间的另一个差别是 JCheckBoxMenuItem 的实例不可以以助记符键为参数来构造。然而, 因为 JCheckBoxMenuItem 最终是从 AbstractButton 中派生出来的, 所以, 可以在构造后为复选框菜单项设置助记符键, 如图 10-8 所示。

2. 初始化和状态

```

protected void init (String, Icon)
public synchronized Object [] getSelectedObjects ()
public boolean getState ()
public synchronized void setState (boolean)

```

除带图标复选框菜单项的文本的垂直位置设置为 SwingConstants.BOTTOM 外, JCheckBoxMenuItem 的 init 方法几乎与 JMenuItem 的 init 方法完全相同。如果缺省的行为不能满足需要, 则 JCheckBoxMenuItem 的扩展可以重载 init 方法。

getSelectedObjects 方法返回一个 Object 数组, 这个数组只有一项。如果选取了复选框菜单项, 则数组中的项就是这个菜单项的文本, 否则, 如果没有选取菜单项, 则数组中的项是 null。

JCheckBoxMenuItem API 提供设置和获取菜单项选取状态的访问方法。

3. 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()
public void setUI (CheckBoxMenuItemUI)
```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

10.4.4 AWT 兼容

表 10-2 列出了由 java.awt.CheckBoxMenuItem 实现的 public 方法和由 JCheckBoxMenuItem 实现的对应方法。

表 10-2 java.awt.CheckBoxMenuItem 的方法和 JCheckBoxMenuItem 的对应方法

java.awt.CheckboxMenuItem 的方法	JCheckBoxMenuItem 的对应方法
void addItemListener (ItemListener)	void addItemListener (ItemListener)
Object [] getSelectedObjects ()	Object [] getSelectedObjects ()
boolean getState ()	boolean getState ()
String paramString ()	String paramString ()
void removeItemListener (ItemListener)	void removeItemListener (ItemListener)
void setState (boolean)	void setState (boolean)

JCheckBoxMenuItem 与 java.awt.CheckboxMenuItem 的源代码完全兼容。JCheckBoxMenuItem API 提供 setState、getState 和 getSelectedObjects 方法来维护与 AWT 复选框菜单项的兼容。

10.5 JRadioButtonMenuItem

Swing 单选按钮菜单项与复选框菜单项相似，它们都是 JMenuItem 的简单扩展。
Swing 复选框菜单项与单选按钮项有两点主要差别。

首先，虽然单选按钮菜单项可以选取或取消选取，但是 JRadioButtonMenuItem 类没有提供获得和设置这种菜单项的选取状态的方法。换句话说，JRadioButtonMenuItem 没有像 JCheckBoxMenuItem 那样提供 setState 方法和 getState 方法。setState 方法和 getState 方法被 JCheckBoxMenuItem 实现以便维护与 AWT 的 CheckboxMenuItem 类的兼容。因为 AWT 没有提供与 Swing 的 JRadioButtonMenuItem 类似的组件，所以不需要这些方法来维护与 AWT 的兼容。

其次，与 JCheckBoxMenuItem 不同，getSelectedObjects 方法不能被 JRadioButtonMenuItem 类重载。因此，为 JRadioButtonMenuItem 的实例调用 getSelectedObjects 将导致对 AbstractButton.getSelectedObjects () 的调用，AbstractButton.getSelectedObjects 方法将返回 null。

图 10-10 示出了创建 JRadioButtonMenuItem 的三个实例的小应用程序。

例 10-8 列出了图 10-10 示出的小应用程序的代码。

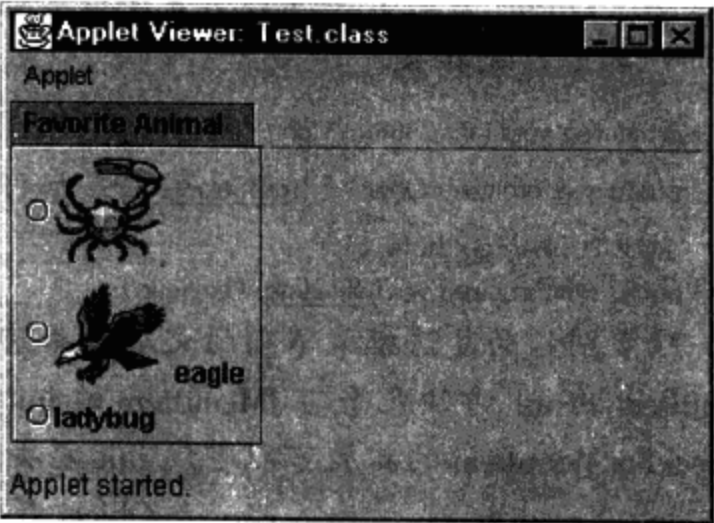


图 10-10 使用单选按钮菜单项

例 10-8 实例化单选按钮菜单项

```

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();

        JMenuBar mb = new JMenuBar ();
        JMenu radioMenu = new JMenu ("Favorite Animal");

        ImageIcon crabIcon = new ImageIcon ("crab.gif");
        ImageIcon eagleIcon = new ImageIcon ("eagle.gif");

        final JMenuItem
            crabItem = new JRadioButtonMenuItem (crabIcon),
            eagleItem = new JRadioButtonMenuItem ("eagle", eagleIcon),
            ladybugItem = new JRadioButtonMenuItem ("ladybug");

        radioMenu.add (crabItem);
        radioMenu.add (eagleItem);
        radioMenu.add (ladybugItem);

        mb.add (radioMenu);
        setJMenuBar (mb);
    }
}

```

如果运行上面所列的小应用程序，就会注意到可以同时选取数个单选按钮菜单项。与 Swing 单选按钮一样，单选按钮菜单项不呈现单选的行为（如果需要单选，则必须把它们添加到一个按钮组中）。与 Swing 单选按钮不同，已创建的不带文本的单选按钮菜单项用一个选取控制来绘制（参见 9.4 节“单选按钮”）。组件总结 10-3 总结了 JRadioButtonMenuItem 类。

组件总结 10-3 JRadioButtonMenuItem

模型：	JToggleButton.ToggleButtonModel
UI 代表：	swing.plaf.basic.BasicRadioButtonMenuItemUI
绘制器：	——
编辑器：	——
激发的事件：	ActionEvent、ChangeEvent、ItemEvent
替换：	——
类图：	参见图 10-6

10.5.1 JRadioButtonMenuItem 属性

JRadioButtonMenuItem 没有添加它自己的新属性，所有的属性都是继承来的。

10.5.2 JRadioButtonMenuItem 事件

与所有的 Swing 按钮一样，激活单选按钮菜单项时将激发动作事件，在修改它们的相关属性时将激发属性变化事件，在它们的状态变化时将激发变化事件。有关由按钮激发的动作事件和

变化事件的更多信息，请参见 8.4.2 节“JButton 事件”，有关属性变化事件的更多信息，请参见 3.2.4 节。另外，单选钮菜单项在它们被选取或取消选取时将激发项事件。有关处理项事件的更多信息，请参见 9.1.2 节“JToggleButton 事件”。

图 10-11 显示了一个小应用程序，它监视由 JRadioButtonMenuItem 的实例激发的事件。

这个小应用程序创建了 JRadioButtonMenuItem 的三个实例并把这些实例添加到菜单中。另外，把每个菜单项都添加到一个按钮组中以确保只能选取一个菜单项。

每个菜单项都配备了三个监听器：ItemListener、ChangeListener 和 ActionListener。

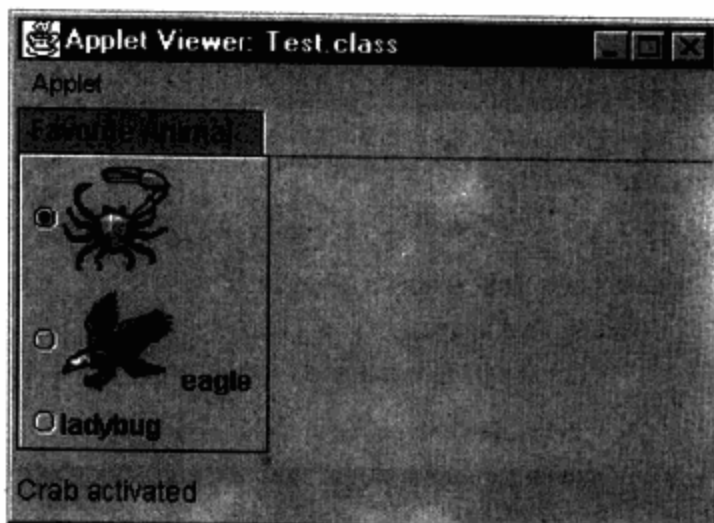


图 10-11 处理 JRadioButtonMenuItem 的实例的

```
public class Test extends JApplet {
    private ImageIcon crabIcon = new ImageIcon ("crab.gif",
                                                "Crab");
    private ImageIcon eagleIcon = new ImageIcon ("eagle.gif",
                                                "Eagle");
    private JMenuItem
        crabItem = new JRadioButtonMenuItem (crabIcon),
        eagleItem = new JRadioButtonMenuItem ("eagle",
                                                eagleIcon),
        ladybugItem = new JRadioButtonMenuItem ("ladybug");
    public void init () {
        ...
        AnItemListener itemListener = new AnItemListener ();
        AnActionListener actionListener = new AnActionListener ();
        AChangeListener changeListener = new AChangeListener ();
        radioMenu.add (crabItem);
        radioMenu.add (eagleItem);
        radioMenu.add (ladybugItem);
        ButtonGroup group = new ButtonGroup ();
        group.add (crabItem);
        group.add (eagleItem);
        group.add (ladybugItem);
        ...
        crabItem.addItemListener (itemListener);
        eagleItem.addItemListener (itemListener);
        ladybugItem.addItemListener (itemListener);
        crabItem.addActionListener (actionListener);
        eagleItem.addActionListener (actionListener);
        ladybugItem.addActionListener (actionListener);
        crabItem.addChangeListener (changeListener);
        eagleItem.addChangeListener (changeListener);
        ladybugItem.addChangeListener (changeListener);
    }
    ...
}
```

这些监听器作为这个小应用程序的内部类实现，以便它们可以更新这个小应用程序的状态区。动作监听器和子项监听器都在这个小应用程序的状态区中显示有关事件的信息。

```
...
class AnActionListener implements ActionListener {
    public void actionPerformed ( ActionEvent e ) {
        JRadioButtonMenuItem item = (JRadioButtonMenuItem )
                                                    e.getSource ();

        String s = getItemDescription (item);
        showStatus (s + " activated");
    }
};

class AnItemListener implements ItemListener {
    public void itemStateChanged ( ItemEvent e ) {
        JRadioButtonMenuItem item = (JRadioButtonMenuItem )
                                                    e.getSource ();

        String s = getItemDescription (item);

        if (e.getStateChange () == ItemEvent.SELECTED)
            s += " selected";
        else
            s += " deselected";

        showStatus (s);
    }
};
...
```

通过调用这个小应用程序的 `getItemDescription` 方法，这些监听器获得激发事件的菜单项的一个描述，`getItemDescription` 方法返回菜单项的文本或菜单项的图标的一个描述（如果这个菜单项有一个图标的话）。

为了说明这样一个事实：`getSelectedObjects()` 总是为 `JRadioButtonMenuItem` 的实例返回 `null`，一个变化监听器调用了 `getSelectedObjects()` 并显示结果。另外，当激发事件的菜单项处于待命状态时，这个变化监听器将在小应用程序的状态区中显示一个消息。

```
...
class AChangeListener implements ChangeListener {
    public void stateChanged ( ChangeEvent e ) {
        Object [] selectedObjs =
            ladybugItem.getSelectedObjects ();

        if (selectedObjs == null)
            System.out.println ("selected objs is null");
        else
            System.out.println (selectedObjs [0] + "selected");

        JRadioButtonMenuItem item =
            (JRadioButtonMenuItem ) e.getSource ();
        if (item.isArmed ()) {
            String s = getItemDescription (item);
            showStatus (s + " armed");
        }
    }
};
...
```

例 10-9 列出了图 10-11 所示的小应用程序的代码。

例 10-9 为 JRadioButton 的实例监视事件

```
import javax.swing.*.*;
import javax.swing.event.*;
import java.awt.*.*;
import java.awt.event.*;

public class Test extends JApplet {
    private ImageIcon crabIcon = new ImageIcon ("crab.gif", "Crab");
    private ImageIcon eagleIcon = new ImageIcon ("eagle.gif", "Eagle");
    private JMenuItem
        crabItem = new JRadioButtonMenuItem (crabIcon),
        eagleItem = new JRadioButtonMenuItem ("eagle", eagleIcon),
        ladybugItem = new JRadioButtonMenuItem ("ladybug");

    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu radioMenu = new JMenu ("Favorite Animal");

        AnItemListener itemListener = new AnItemListener ();
        AnActionListener actionListener = new AnActionListener ();
        AChangeListener changeListener = new AChangeListener ();

        radioMenu.add (crabItem);
        radioMenu.add (eagleItem);
        radioMenu.add (ladybugItem);

        ButtonGroup group = new ButtonGroup ();
        group.add (crabItem);
        group.add (eagleItem);
        group.add (ladybugItem);

        mb.add (radioMenu);
        setJMenuBar (mb);

        crabItem.addItemListener (itemListener);
        eagleItem.addItemListener (itemListener);
        ladybugItem.addItemListener (itemListener);

        crabItem.addActionListener (actionListener);
        eagleItem.addActionListener (actionListener);
        ladybugItem.addActionListener (actionListener);

        crabItem.addChangeListener (changeListener);
        eagleItem.addChangeListener (changeListener);
        ladybugItem.addChangeListener (changeListener);
    }

    private String getItemDescription (
        JRadioButtonMenuItem item) {
        String s;
        ImageIcon icon = (ImageIcon) item.getIcon ();
        if (icon != null) return icon.getDescription ();
        else return item.getText ();
    }

    // Inner class event handlers follow ...

    class AnActionListener implements ActionListener {
        public void actionPerformed (ActionEvent e) {
```

```

        JRadioButtonMenuItem item = (JRadioButtonMenuItem)
            e.getSource ();
        String s = getItemDescription (item);
        showStatus (s + " activated");
    }
};

class AChangeListener implements ChangeListener {
    public void stateChanged (ChangeEvent e) {
        Object [] selectedObjs =
            ladybugItem.getSelectedObjects ();

        if (selectedObjs == null)
            System.out.println ("selected objs is null");
        else
            System.out.println (selectedObjs [0] + "selected");

        JRadioButtonMenuItem item =
            (JRadioButtonMenuItem) e.getSource ();

        if (item.isArmed ()) {
            String s = getItemDescription (item);
            showStatus (s + " armed");
        }
    }
};

class AnItemListener implements ItemListener {
    public void itemStateChanged (ItemEvent e) {
        JRadioButtonMenuItem item = (JRadioButtonMenuItem) e.getSource ();
        String s = getItemDescription (item);

        if (e.getStateChange () == ItemEvent.SELECTED)
            s += " selected";
        else
            s += " deselected";
        showStatus (s);
    }
};
}

```

10.5.3 JRadioButtonMenuItem 类总结

类总结 10-4 列出了 JRadioButtonMenuItem 的 public 和 protected 的变量和方法。

类总结 10-4 JRadioButtonMenuItem

扩展: JMenuItem

实现: javax.accessibility.Accessible

1. 构造方法

```

public JRadioButtonMenuItem ()
public JRadioButtonMenuItem (String)
public JRadioButtonMenuItem (String, boolean selected)
public JRadioButtonMenuItem (String, Icon)
public JRadioButtonMenuItem (String, Icon, boolean selected)
public JRadioButtonMenuItem (Icon)
public JRadioButtonMenuItem (Icon, boolean selected)

```

可以用上面所列的这些构造方法来指定在菜单项中显示的字符串、图标及选取状态。

2. 方法

(1) 初始化/请求焦点

```
protected void init (String, Icon)
public void requestFocus ()
```

JRadioButtonMenuItem.init 方法与 JCheckBoxMenuItem 的 init 方法相同。如果需要，则扩展可以重载 init 方法。与 JCheckBoxMenuItem 一样，JRadioButtonMenuItem 的实例不捕获焦点，这是通过用一个空实现来重载 requestFocus () 来保证的。

(2) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()
public void setUI (RadioButtonMenuItemUI)
```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

10.5.4 AWT 兼容

AWT 没有提供与 Swing 的 JRadioButtonMenuItem 类似的组件。

Swing 提示

JCheckBoxMenuItem API 与 JRadioButtonMenuItem API 的比较

初看起来，JCheckBoxMenuItem API 与 JRadioButtonMenuItem API 几乎完全相同。然而，复选框菜单项与单选按钮菜单项唯一显著的区别是指示选取的控制是不同的。

JCheckBoxMenuItem 的 API 不同于 JRadioButtonMenuItem 的 API，因为 JCheckBoxMenuItem 维护与 AWT 复选框菜单项的 API 的兼容性。例如，JCheckBoxMenuItem 提供 setState 和 getState 方法来维护与 java.awt.CheckboxMenuItem 的兼容性。相反，JRadioButtonMenuItem 没有与之类似的 AWT 组件，因此，在它的 API 中没有包括那些维护与一个 AWT 类兼容的方法。

10.6 JMenu

如前所述，Swing 菜单本质上是按钮，它有与之相关联的弹出式菜单。当激活一个菜单时，它的弹出式菜单在这个菜单的下面显示。

图 10-12 示出了一个带有“File”菜单的小应用程序。“File”菜单配备了一个图标，而且这个菜单中还添加了一个按钮。可以操纵已添加到菜单中的组件，如图 10-12 右图所示，它显示菜单中正在被激活的按钮。

例 10-10 列出了图 10-12 所示的小应用程序的代码。

例 10-10 JMenu 的一个简单例子

```
import javax.swing.*;
import java.awt.*;
```

```

import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");

        fileMenu.add ("New...");
        fileMenu.add ("Open...");
        fileMenu.add ("Save...");
        fileMenu.add ("Save As...");
        fileMenu.addSeparator ();
        fileMenu.add ("Exit");

        fileMenu.add (new JButton ("a button"));
        fileMenu.setIcon (new ImageIcon ("disk.gif"));

        mb.add (fileMenu);
        setJMenuBar (mb);
    }
}

```

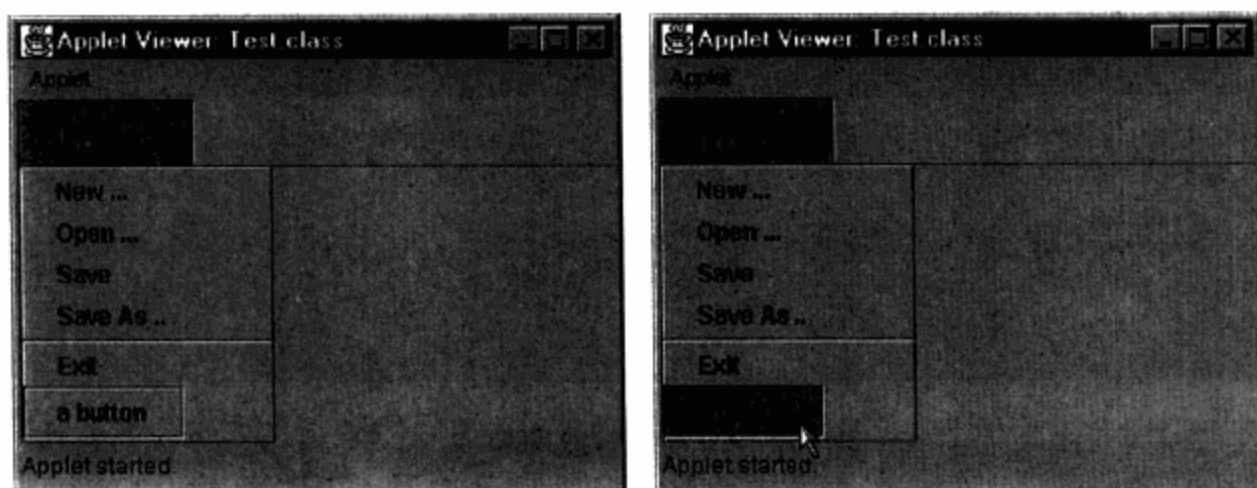


图 10-12 有一个图标和一个组件的菜单

因为 Swing 菜单是按钮，所以可以为 JMenu 的一个实例设置一个图标，如例 10-10 所示。Swing 菜单还可以把任意一个组件添加到它们中，如图 10-12 所示，JButton 的实例被添加到 File 菜单中。

注意 在 Swing 1.1 FCS 中，当操纵包含在菜单中的组件以便改变这个组件的大小时，Swing 菜单不能正确地重新绘制。例如，如果把一个树添加到一个菜单中，当树的文件夹被展开或折叠时，将不能正确地重新调整这个菜单的大小。

10.6.1 动态修改菜单

JMenu 提供了许多方法来添加菜单/插入菜单/删除菜单中的菜单项/进入菜单/退出菜单。添加或插入动作或菜单项的方法返回对最后添加或插入的菜单项的一个引用。添加或插入一个字符串的方法不返回对菜单项的一个引用；然而，把一个字符串添加到一个菜单中将导致菜单项的创建，这个菜单项将添加到这个菜单中。

除了把菜单项添加到菜单中外，组件和分隔线也可以添加到 JMenu 的实例中。把一个组件或分隔线添加到菜单中不会导致菜单项的创建，组件或分隔线可简单地插入到菜单的弹出式菜

单中。

可以在任何 JMenu 实例中添加和删除菜单监听器。有关菜单监听器的更多信息，请参见 10.6.4 节“JMenu 事件”。

图 10-13 所示的小应用程序除图解说明了启用和禁用菜单项外，还图解说明了添加和删除菜单中的菜单项。这个小应用程序还图解说明了为一个菜单项设置字体。

这个小应用程序实现 JMenu 的一个扩展 (SelfModifyingMenu)，它包含图 10-13 所示的菜单项。SelfModifyingMenu 类添加菜单项和分隔线，并把一个动作监听器添加到这些菜单项中。

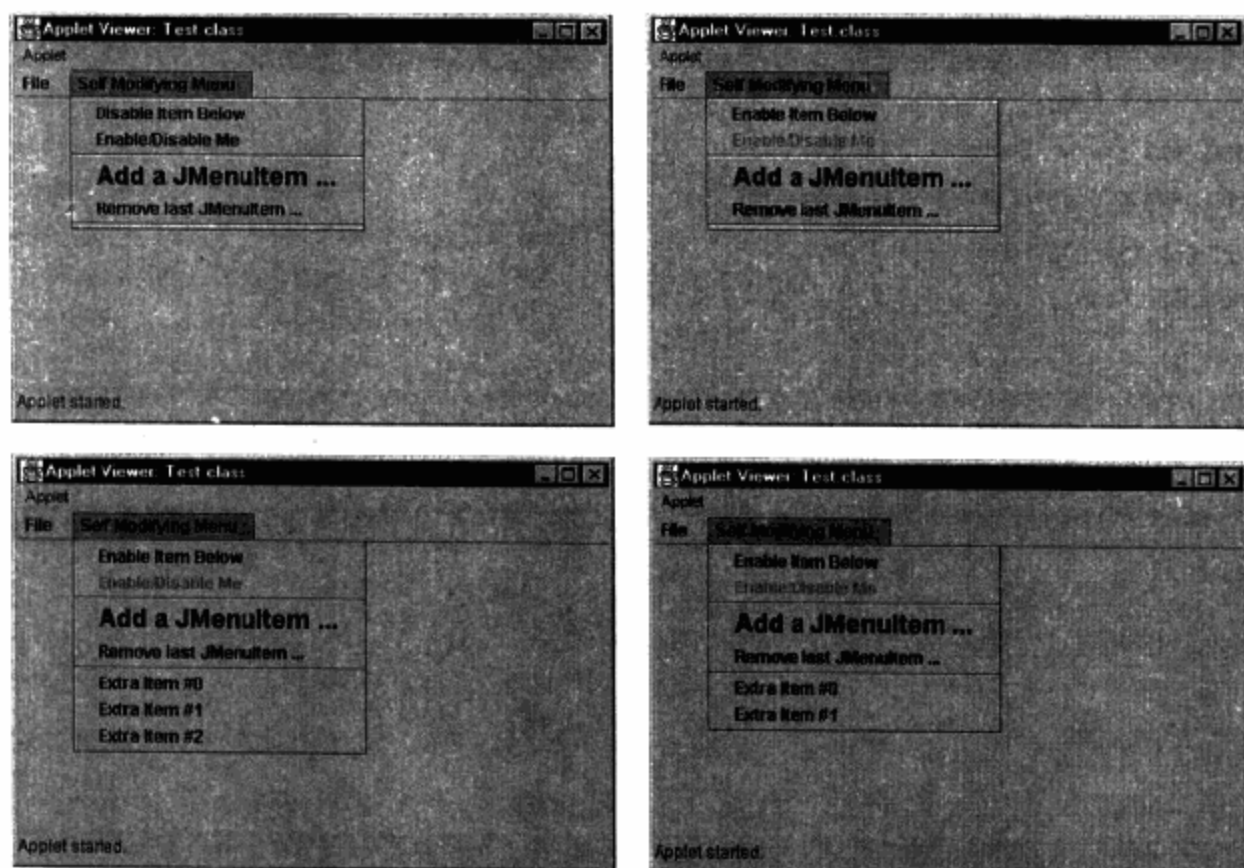


图 10-13 动态修改菜单项

```
class SelfModifyingMenu extends JMenu {
    private Vector newItem = new Vector ();
    private Listener menuItemListener = new Listener ();
    private JMenuItem toggleItem, enablerItem,
                        addItem, removeItem;

    public SelfModifyingMenu () {
        super ("Self Modifying Menu"); //set menu title

        add (enablerItem = new JMenuItem ("Disable Item Below"));
        add (toggleItem = new JMenuItem ("Enable/Disable Me"));
        addSeparator ();

        add (addItem = new JMenuItem ("Add a JMenuItem ..."));
        add (removeItem = new JMenuItem ("Remove last JMenuItem..."));
        addItem.setFont (new Font ("Helvetica", Font.BOLD, 18));
        addSeparator ();

        enablerItem.addActionListener (menuItemListener);
        toggleItem.addActionListener (menuItemListener);
        addItem.addActionListener (menuItemListener);
        removeItem.addActionListener (menuItemListener);
    }
}
```

当选取了一个菜单项时，动作监听器确定哪个菜单项是这个事件的源，并据此进行动作。如果激活的菜单项是 `enablerItem`，则调用这个小应用程序的 `toggleItem` 方法（该方法只触发 `toggleItem` 的启用状态），而且更新 `enablerItem` 的文本。如果激活的菜单项是 `addItem`，则调用这个小应用程序的 `addItem` 方法（该方法把一个菜单项添加到这个菜单中）。如果激活的菜单项是 `removeItem`，则调用这个小应用程序的 `removeItem` 方法，该方法删除最后添加到这个菜单中的菜单项。

```
...
public void addItem () {
    JMenuItem newItem =
        new JMenuItem ("Extra Item #" + newItems.size ());

    add (newItem);
    newItems.addElement (newItem);
}
public void removeLastItem () {
    if (newItems.size () == 0)
        System.out.println ("Nothing to remove!");
    else {
        JMenuItem removeMe =
            (JMenuItem) newItems.lastElement ();

        remove (removeMe);
        newItems.removeElement (removeMe);
    }
}
public void toggleItem () {
    if (toggleItem.isEnabled ()) toggleItem.setEnabled (false);
    else toggleItem.setEnabled (true);
}
class Listener implements ActionListener {
    public void actionPerformed (ActionEvent event) {
        JMenuItem item = (JMenuItem) event.getSource ();

        if (item == enablerItem) {
            toggleItem ();

            if (toggleItem.isEnabled ())
                enablerItem.setText ("Disable Item Below");
            else
                enablerItem.setText ("Enable Item Below");
        }
        else if (item == addItem) addItem ();
        else if (item == removeItem) removeLastItem ();
    }
}
```

例 10-11 列出了图 10-13 所示的小应用程序的完整代码。

例 10-11 一个自修改菜单

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
```

```

public class Test extends JApplet {
    private SelfModifyingMenu selfModifyingMenu;

    public void init () {
        JMenuBar menuBar = To ();
        createMenus (menuBar);
        setJMenuBar (menuBar);
    }

    public void createMenus (JMenuBar mbar) {
        mbar.add (createFileMenu ());
        mbar.add (selfModifyingMenu = new SelfModifyingMenu ());
    }

    private JMenu createFileMenu () {
        JMenu fileMenu = new JMenu ("File");
        JMenuItem quitItem = new JMenuItem ("Quit");

        fileMenu.add (quitItem);

        quitItem.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent event) {
                System.exit (0);
            }
        })
        return fileMenu;
    }

    class SelfModifyingMenu extends JMenu {
        private Vector newItems = new Vector ();
        private Listener menuItemListener = new Listener ();
        private JMenuItem toggleItem, enablerItem,
            addItem, removeItem;

        public SelfModifyingMenu () {
            super ("Self Modifying Menu");

            add (enablerItem = new JMenuItem ("Disable Item Below"));
            add (toggleItem = new JMenuItem ("Enable/Disable Me"));
            addSeparator ();

            add (addItem = new JMenuItem ("Add a JMenuItem ..."));
            add (removeItem = new JMenuItem ("Remove last JMenuItem ..."));
            addItem.setFont (new Font ("Helvetica", Font. BOLD, 18));
            addSeparator ();

            enablerItem.addActionListener (menuItemListener);
            toggleItem.addActionListener (menuItemListener);
            addItem.addActionListener (menuItemListener);
            removeItem.addActionListener (menuItemListener);
        }

        public void addItem () {
            JMenuItem newItem =
                new JMenuItem ("Extra Item #" + newItems.size ());

            add (newItem);
            newItems.addElement (newItem);
        }

        public void removeLastItem () {
            if (newItems.size () == 0)
                System.out.println ("Nothing to remove!");
        }
    }
}

```

```

else {
    JMenuItem removeMe =
        (JMenuItem) newItems.lastElement ();
    remove (removeMe);
    newItems.removeElement (removeMe);
}
}
public void toggleItem () {
    if (toggleItem.isEnabled ()) toggleItem.setEnabled (false);
    else toggleItem.setEnabled (true);
}
class Listener implements ActionListener {
    public void actionPerformed (ActionEvent event) {
        JMenuItem item = (JMenuItem) event.getSource ();
        if (item == enablerItem) {
            toggleItem ();
            if (toggleItem.isEnabled ())
                enablerItem.setText ("Disable Item Below");
            else
                enablerItem.setText ("Enable Item Below");
        }
        else if (item == addItem) addItem ();
        else if (item == removeItem) removeLastItem ();
    }
}

```

10.6.2 右拉式菜单

JMenu 扩展 JMenuItem, 因此, 用 Swing 来实现右拉式菜单是很容易的。图 10-14 所示的小应用程序创建了具有三层结构的层叠菜单。

例 10-12 列出了图 10-14 所示的小应用程序的代码。

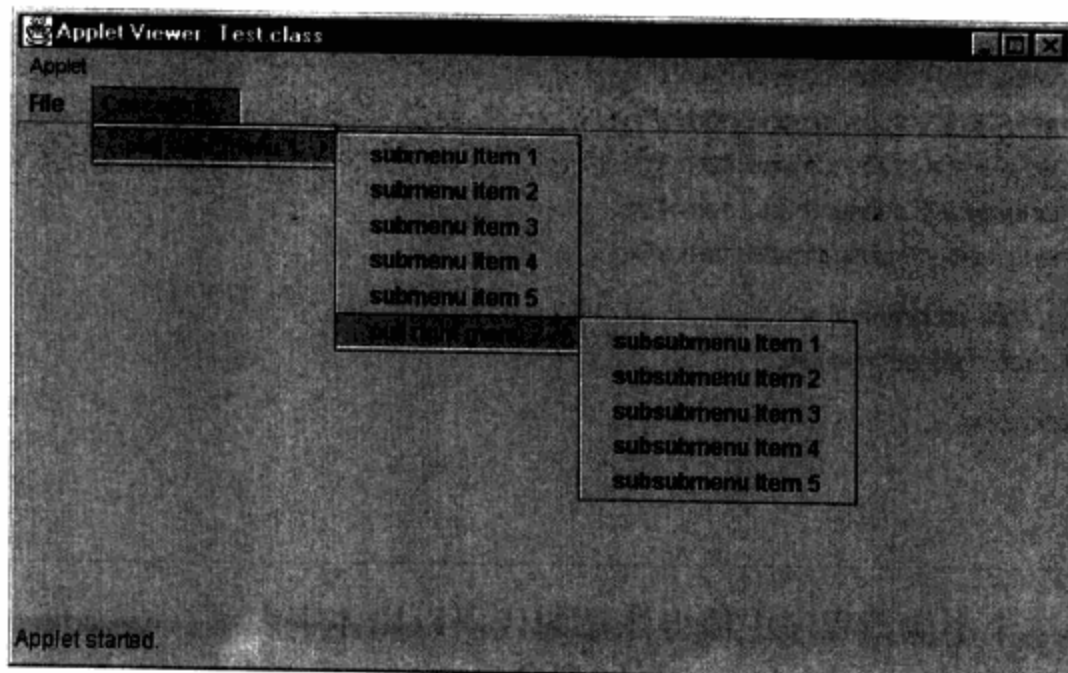


图 10-14 右拉式菜单

例 10-12 右拉式菜单

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {

    public void init () {
        JMenuBar menuBar = new JMenuBar ();
        createMenus (menuBar);
        setJMenuBar (menuBar);
    }

    public void createMenus (JMenuBar mbar) {
        mbar.add (createFileMenu ());
        mbar.add (createCascadingMenu ());
    }

    private JMenu createFileMenu () {
        JMenu fileMenu = new JMenu ("File");
        JMenuItem quitItem = new JMenuItem ("Quit");
        fileMenu.add (quitItem);

        quitItem.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent event) {
                System.exit (0);
            }
        });
        return fileMenu;
    }

    private JMenu createCascadingMenu () {
        JMenu cascading = new JMenu ("Cascading");
        JMenu submenu = new JMenu ("pull right menu 1");
        JMenu subsubmenu = new JMenu ("pull right menu 2");

        submenu.add ("submenu item 1");
        submenu.add ("submenu item 2");
        submenu.add ("submenu item 3");
        submenu.add ("submenu item 4");
        submenu.add ("submenu item 5");

        subsubmenu.add ("subsubmenu item 1");
        subsubmenu.add ("subsubmenu item 2");
        subsubmenu.add ("subsubmenu item 3");
        subsubmenu.add ("subsubmenu item 4");
        subsubmenu.add ("subsubmenu item 5");

        cascading.add (submenu);
        submenu.add (subsubmenu);

        return cascading;
    }
}
```

上面所列的这个小程序中值得注意的代码以黑体字表示。cascading 菜单把 submenu 添加到它自身中，submenu 把 subsubmenu 添加到自身中。

组件总结 10-4 总结了 JMenu 类。

组件总结 10-4 JMenu

模型: ButtonModel
UI 代表: javax.swing.plaf.basic.BasicMenuUI
绘制器: _____
编辑器: _____
激发的事件: MenuEvent
替换: java.awt.Menu
类图:

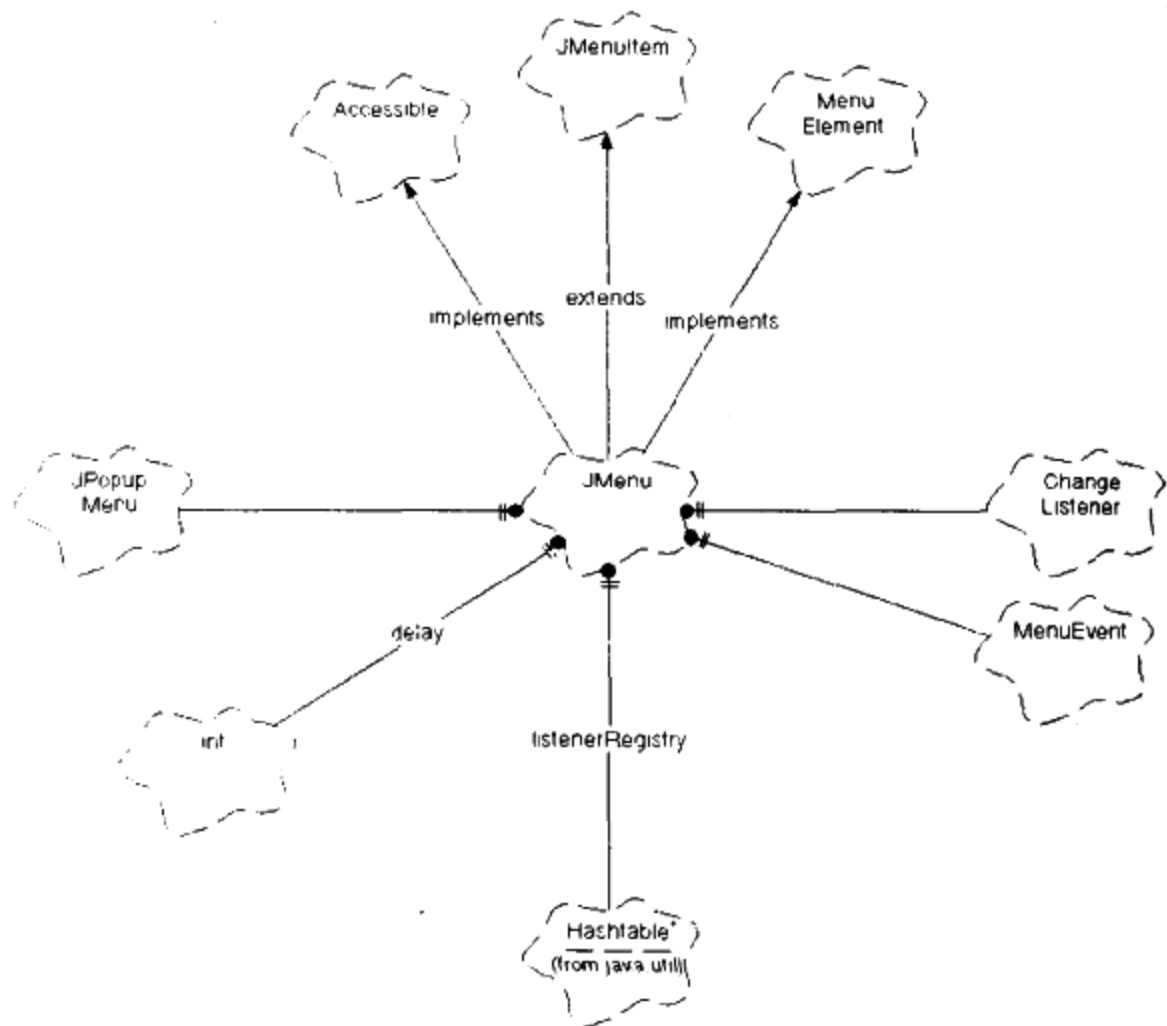


图 10-15 JMenu 的类图

JMenu 扩展 JMenuItem，并且实现 Accessible 接口和 MenuElement 接口。因为 JMenu 扩展 JMenuItem，所以 JMenu 的实例可以被添加到菜单中，导致具有了创建层叠菜单的功能。另外，因为 JMenu 扩展 JMenuItem，JMenuItem 又扩展 AbstractButton，所以菜单是按钮而且可以被操纵。

每个 JMenu 实例维护对一个 JPopupMenu 实例的一个 private 引用（当激活菜单时将显示的一个弹出式菜单）。菜单还维护一个 private 哈希表，这个表被用于在监听器被删除时清除对监听器的引用，以便这个监听器可以被垃圾箱收集。

JMenu 还维护对 MenuEvent 和 ChangeListener 实例的 Private 引用。当一个菜单事件发生时，这个菜单事件被传送给菜单监听器，而且变化监听器监听这个菜单模型的变化。

最后，菜单维护一个 integer 值，这个值代表菜单被选取和与之相关联的弹出式菜单被显示之间的时间延迟（以毫秒为单位）。缺省时，延迟是 0 毫秒，JMenu 属性参见表 10-3。

10.6.3 JMenu 属性

表 10-3 列出了与 JMenu 的实例相关联的属性。

表 10-3 JMenu 属性

属性名	数据类型	访问 ^②	类型 ^③	缺省 ^④
delay	int	SG	S	0
item	JMenuItem	G	I	——
itemCount	int	G	S	——
menuComponent	Component	G	I	——
menuComponentCount	int	G	S	——
menuComponents	int	G	S	——
menuLocation	Point	S	S	L&F
model	ButtonModel	SG	B	ButtonModel
popupMenu	JPopupMenu	G	S	JPopupMenu
popupMenuVisible	boolean	SG	S	false
tearOff ^⑤	boolean	CG	S	false
topLevelMenu	boolean	G	S	——

① 在 Swing 1.1 FCS 中没有实现
② C = 可在构造时刻设置/G = 获取方法/S = 设置方法
③ B = 关联的（激发 PropertyChangeEvent）/C = 受约束的/I = 索引的/S = 简单的
④ L&F = 与界面样式有关

delay—— 一个延迟（以毫秒为单位），在选取菜单和显示它的弹出式菜单之间的时间。delay 属性是为菜单的 UI 代表提供的；菜单组件本身不做任何与这个属性有关的事情。对 Swing 1.1 FCS 而言，没有一个 Swing 的界面样式使用 delay 属性。

item—— 一个索引属性，它提供对菜单的菜单项的访问。代表一个菜单项的索引的 integer 值被传送给 JMenu.getItem 方法。如果在指定的索引中没有菜单项，则 getItem 方法返回对菜单本身的一个引用。注意：getItem 方法只提供对包含在一个菜单中的菜单项的访问。如果一个菜单包含其他类型的组件，则可以通过 menuComponent 属性来访问这些组件。

itemCount—— 一个菜单所包含的组件数。提供 itemCount 属性是为了向后兼容 java.awt.Menu。除非需要与 AWT 菜单兼容，否则使用 menuComponentCount 属性，它等价于 itemCount 属性。

menuComponent—— 一个索引属性，它提供对菜单弹出式菜单所包含的组件（包括菜单项）的访问。已索引的 item 属性只提供对菜单项的访问，而 menuComponent 属性却提供对一个菜单的弹出式菜单所包含的所有组件的访问。

menuComponentCount—— 代表一个菜单的弹出式菜单中所包含的组件数。

menuComponents—— 一个组件数组，这些组件包含在一个菜单的弹出式菜单中。

menuLocation—— 一个只写属性，它代表一个菜单的弹出式菜单的左上角。

model—— 菜单的模型是 ButtonModel 类型，JMenu 提供设置和获取方法来对它的模型进行访问。

popupMenu—— 与一个 JMenu 实例相关联的弹出式菜单。PopupMenu 属性是只读的。

popupMenuVisible—— 与一个 JMenu 实例相关联的弹出式菜单的可见性。popupMenuVisible 属性是只读的。

tearOff—— 指示一个菜单是否可以“移开”，并放在一个单独的窗口中。Swing 1.1 FCS 不支持 tearOff 属性。

topLevelMenu—— 一个 boolean 属性，它指示一个菜单是否是一个顶层菜单。顶层菜单在

10.2 节“菜单和弹出式菜单”中介绍。

注意 下面的 JMenu 属性在 Swing 1.1 FCS 中不能使用：

- delay (延迟)[⊖]。
- menu Location (菜单位置)。
- pupup size (弹出式菜单的大小)。
- tear off (移开)。

10.6.4 JMenu 事件

在选取、取消选取或取消 JMenu 的实例时将激发菜单事件。点击菜单的弹出式菜单以外的区域，可以取消一个菜单。为了监听菜单选取事件，在 swing.event 包中定义了一个监听器接口。

接口总结 10-2 MenuListener

```
public abstract void menuCanceled (MenuEvent)
public abstract void menuDeselected (MenuEvent)
public abstract void menuSelected (MenuEvent)
```

在 Swing 1.1 FCS 中，JMenu 类不会激活菜单取消事件。换句话说，与 JMenu 的一个实例相关联的菜单监听器将不会调用 menuCanceled 方法。

例 10-13 所列的小应用程序把一个菜单监听器添加到 JMenu 的一个实例中，并在取消一个在 MenuListener 接口中的方法时显示一个消息。

例 10-13 监听菜单事件

```
import javax.swing.*.*;
import javax.swing.event.*;
import java.awt.*.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");

        fileMenu.add ("New ...");
        fileMenu.add ("Open ...");
        fileMenu.add ("Save");
        fileMenu.add ("Save As ...");
        fileMenu.addSeparator ();
        fileMenu.add ("Exit");

        mb.add (fileMenu);
        setJMenuBar (mb);

        fileMenu.addMenuListener (new MenuListener () {
            public void menuCanceled (MenuEvent e) {
                System.out.println ("menu canceled");
            }
            public void menuSelected (MenuEvent e) {
                System.out.println ("menu selected");
            }
        });
    }
}
```

[⊖] 根据界面样式来设置延迟；缺省界面样式都不使用这个属性。

```

        public void menuDeselected (MenuEvent e) {
            System.out.println ("menu deselected");
        }
    }
}

```

注意 MenuEvent 类（它扩展 java.util.EventObject 类）不提供它自己的方法。从 MenuEvent 的一个实例中可以获得的唯一信息是事件源。

10.6.5 JMenu 类总结

类总结 10-5 列出了 JMenu 的 public 和 protected 变量和方法。

类总结 10-5 JMenu

扩展：JMenuItem

实现：MenuElement、javax.accessibility.Accessible

1. 构造方法

```

public JMenu ()
public JMenu (String)
public JMenu (String, boolean isTearOff)

```

JMenu 提供了一个无参数构造方法，它创建一个没有文本的菜单。可以用这个无参数构造方法来构造菜单，在这个菜单创建后再确定这个菜单的标题。

JMenu 提供了两个带一个字符串参数的构造方法。字符串参数是将在菜单中显示的文本，而上面所列的第三个构造方法中指定的 boolean 值确定菜单是否是一个移动菜单。在 Swing 1.1FCS 中没有实现移动菜单。

2. 方法

(1) 监听器

```

public void addMenuListener (MenuListener)
public void removeMenuListener (MenuListener)

```

JMenu 类提供添加和删除菜单监听器的方法。有关 MenuListener 类的详细内容，请参见 10.6.4 节“JMenu 事件”。

(2) 配置

```

public JMenuItem add (Action)
public JMenuItem add (JMenuItem)
public JMenuItem add (String)
public Component add (Component)

public void addSeparator ()

public JMenuItem insert (Action, int index)
public JMenuItem insert (JMenuItem, int index)
public void insert (String, int index)
public void insertSeparator (int index)

public void remove (JMenuItem)
public void remove (int index)
public void removeAll ()

```

配置菜单是一个简单的问题，因为 JMenu 提供了上面所列的这些方法来添加、插入和删除对象。可以把菜单项、字符串、组件、动作和分隔线添加到一个菜单中或从一个菜单中删除。

因为在菜单激活时菜单实际上是显示一个弹出式菜单的按钮，所以上面所列的方法配置菜单的弹出式菜单而不是菜单本身。

上面所列的所有 add... 方法把一个菜单项添加到菜单的尾部。当把动作或字符串添加到一个菜单中时，这个菜单创建一个菜单项，这个菜单项分别从 add (Action) 和 add (String) 中返回。Add (JMenuItem) 方法返回对传送给它的菜单项的一个引用。

把一个组件添加到一个菜单中，不会创建一个菜单项，换句话说，add (Component) 不会创建一个菜单项。因为菜单是容器，所以组件直接添加到菜单中。

创建分隔线也不会创建菜单项。addSeparator 方法把一个分隔线 (JSeparator 的一个扩展) 添加到一个菜单的弹出式菜单中。

当动作添加到一个菜单中时，代表一个动作的菜单项根据从这个动作中获得的信息来设置它的文本和图标。有关如何构造一个动作的菜单项的介绍，请参见 5.3 节“动作”。

(3) 弹出式菜单组件

```
public Component getMenuComponent (int index)
public int getMenuComponentCount ()
public Component [] getMenuComponents ()
```

上面所列的这些方法与菜单的弹出式菜单有关。例如，getMenuComponentCount () 返回包含在菜单的弹出式菜单中的组件数。

(4) 属性访问方法

```
public int getDelay ()
public JMenuItem getItem (int index)
public int getItemCount ()
public JPopupMenu getPopupMenu ()
public void setAccelerator (KeyStroke)
public void setDelay (int)
public void setMenuLocation (int x, int y)
public void setModel (ButtonModel)
public void setPopupMenuVisible (boolean)
public void setSelected (boolean)

public boolean isMenuComponent (Component)
public boolean isPopupMenuVisible ()
public boolean isSelected ()
public boolean isTearOff ()
public boolean isTopLevelMenu ()
```

上面所列的这些方法是 JMenu 属性的访问方法。有关这些属性的详细介绍，请参见 10.6.3 节“JMenu 属性”。

图 10-16 所示的小应用程序在按钮激活时，用 getItemCount 和 getItem 方法来显示这个小应用程序菜单中菜单项的信息。

这个小应用程序实现一个类 (MenuBarPrinter)，这个类可以包含任何 Swing 菜单栏中的菜单和菜单项的信息。MenuBarPrinter 实现一个 static print 方法，这个方法使用 JMenuBar.getCount () 和 JMenuBar.getMenu () 来获得菜单栏中的菜单数和菜单栏中的每个菜单。

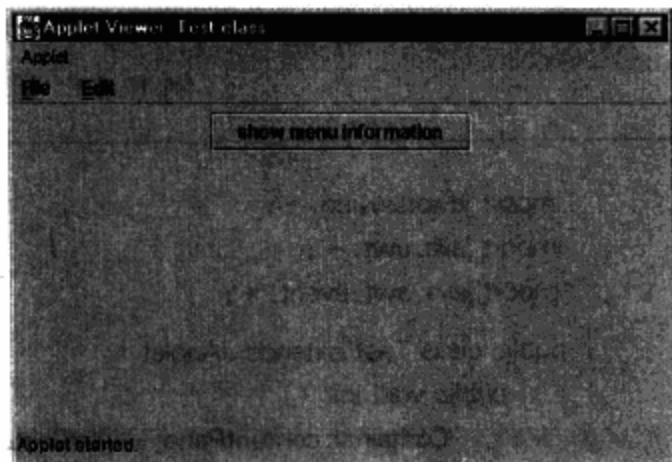


图 10-16 菜单条中菜单的访问信息

```

class MenuBarPrinter {
    static public void print (JMenuBar menubar) {
        int numMenus = menubar.getMenuCount ();
        JMenu nextMenu;

        JMenuItem nextItem;

        System.out.println ();
        System.out.println ("MenuBar has " +
                            menubar.getMenuCount () +
                            " menus");

        System.out.println ();

        for (int i=0; i < numMenus; ++i) {
            nextMenu = menubar.getMenu (i);

            System.out.println (nextMenu.getText () + " menu ...");
            System.out.println (nextMenu);

            int numItems = nextMenu.getItemCount ();

            for (int j=0; j < numItems; ++j) {
                nextItem = nextMenu.getItem (j);
                System.out.println (nextItem);
            }
            System.out.println ();
        }
    }
}

```

小应用程序运行时产生的如下输出：

```

File menu...
javax.swing.JMenu [, 0, 1, 39x19, layout = javax.swing.OverlayLayout, JMenu]
javax.swing.JMenuItem [, 0, 0, 0x0, invalid, layout = javax.swing.overlayLayout]
javax.swing.JMenuItem [, 0, 0, 0x0, invalid, layout = javax.swing.overlayLayout]
javax.swing.JMenuItem [, 0, 0, 0x0, invalid, layout = javax.swing.overlayLayout]
javax.swing.JMenuItem [, 0, 0, 0x0, invalid, layout = javax.swing.overlayLayout]
javax.swing.JMenu [, 0, 1, 39x19, layout = javax.swing.overlayLayout, JMenu]
javax.swing.JMenuItem [, 0, 0, 0x0, invalid, layout = javax.swing.overlayLayout]

Edit menu...
javax.swing.JMenu [, 39, 1, 41x19, layout = javax.swing.overlayLayout, JMenu]
javax.swing.JMenuItem [, 0, 0, 0x0, invalid, layout = javax.swing.overlayLayout]
javax.swing.JMenuItem [, 0, 0, 0x0, invalid, layout = javax.swing.overlayLayout]
javax.swing.JMenuItem [, 0, 0, 0x0, invalid, layout = javax.swing.overlayLayout]

```

例 10-14 列出了图 10-16 示出的小应用程序的完整代码。

例 10-14 显示一个菜单条中菜单的信息

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();

        final JMenuBar mnb = new JMenuBar ();
        final MenuBarPrinter printer = new MenuBarPrinter ();
    }
}

```

```

JMenu fileMenu = new JMenu ("File");
JMenu editMenu = new JMenu ("Edit");
JMenuItem exitItem = new JMenuItem ("Exit");

fileMenu.setMnemonic ('F');
editMenu.setMnemonic ('E');

fileMenu.add ("New ...");
fileMenu.add ("Open ...");
fileMenu.add ("Save");
fileMenu.add ("Save As ...");
fileMenu.addSeparator ();
fileMenu.add (exitItem);

editMenu.add ("Cut");
editMenu.add ("Copy");
editMenu.add ("Paste");

mb.add (fileMenu);
mb.add (editMenu);
setJMenuBar (mb);

JButton button = new JButton ("show menu information");
contentPane.setLayout (new FlowLayout ());
contentPane.add (button);

button.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        printer.print (mb);
    }
});

exitItem.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        System.exit (0);
    }
});
}

class MenuBarPrinter {
    static public void print (JMenuBar menubar) {
        int numMenus = menubar getMenuCount ();
        JMenu nextMenu;

        JMenuItem nextItem;

        System.out.println ();
        System.out.println ("MenuBar has " +
            menubar getMenuCount () +
            " menus");
        System.out.println ();

        for (int i=0; i < numMenus; ++i) {
            nextMenu = menubar getMenu (i);
            System.out.println (nextMenu.getText () + " menu ...");
            System.out.println (nextMenu);

            int numItems = nextMenu.getItemCount ();

            for (int j=0; j < numItems; ++j) {
                nextItem = nextMenu.getItem (j);
                System.out.println (nextItem);
            }
        }
    }
}

```

```
System.out.println ();
```

这个小应用程序为“File”和“Edit”菜单都设置了助记符键。JMenu 重载 setAccelerator 以便抛出一个异常，指出不能在菜单中使用快捷键，只能使用助记符键。

(5) MenuElement 接口

```
public Component getComponent ()
public MenuElement [] getSubElements ()
public void menuSelectionChanged (boolean)
public void processKeyEvent (KeyEvent, MenuElement [], MenuSelectionManager)
public void processMouseEvent (MouseEvent, MenuElement [], MenuSelectionManager)
```

上面所列的这些方法在 MenuElement 接口中定义。有关 MenuElement 接口的更多信息，请参见 10.7 节“菜单元素”。

(6) 监听器/事件激发

```
protected PropertyChangeListener createActionChangeListener (JMenuItem)
protected JMenu.WinListener createWinListener (JPopupMenu)
protected void fireMenuCanceled ()
protected void fireMenuDeselected ()
protected void fireMenuSelected ()
protected void processKeyEvent (KeyEvent)
```

上面所列的这些 protected 方法实现基本的 JMenu 行为，如果这些行为不能满足需要，则可以在扩展中重载这些 protected 方法。通常很少重载这些 protected 方法。

createActionChangeListener 方法创建一个类的一个实例，这个实例实现 PropertyChangeListener 接口以响应已添加到一个菜单中的 Action 的实例的属性变化。缺省情况下，属性变化监听器对动作名、小图标和允许状态属性的变化做出反应。JMenu 的扩展可以重载 createActionChangeListener 方法并安装它们自己的属性变化监听器来响应添加到菜单中的动作的属性变化。有关动作的详细内容，请参见 5.3 节“动作”。

createWinListener 方法创建一个实现 WindowListener 接口的类的一个实例。缺省情况下，当关闭与这个菜单相关联的弹出式菜单时，这个方法创建的窗口监听器只检测这个菜单。JMenu 的扩展可以重载 createWinListener，而且如果需要不同的行为，则安装它们自己的窗口监听器。

当选取或取消选取菜单时，fireMenuSelected 和 fireMenuDeselected 方法分别把菜单事件发送给监听器。重载 processKeyEvent 方法以便忽略多余的键击。

(7) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()
public void setUI (MenuUI)
```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

Swing 提示

Swing 菜单不支持快捷键，只支持助记符键

只能为菜单项设置快捷键，但不能为菜单设置快捷键；应当重载 `JMenu.setAccelerator` 以便弹出一个异常信息。从 10.3.1 节中我们知道，助记符键可以打开菜单和显示这些菜单。相反，快捷键只调用与一个菜单项相关联的动作（快捷键不显示这个菜单）。因为除显示菜单的弹出式菜单外，再没有与菜单相关联的动作，所以，不能为菜单设置快捷键。输入一个与一个菜单相关联的助记符键将显示菜单的弹出式菜单。助记符键把一个键盘捷径键与一个菜单相关联。通过调用 `setMnemonic` 方法来设置助记符键，其中 `setMnemonic` 方法是 `JMenu` 从 `JMenuItem` 从 `AbstractButton` 继承来的。

10.6.6 AWT 兼容

Swing 的菜单与 AWT 的菜单明显不同。Swing 菜单（和菜单项）是按钮，因为它们扩展 `AbstractButton`，因此，可以以操纵任何按钮的方式来操纵它们。而且，由于 `AbstractButton` 扩展 `JComponent`，`JComponent` 又扩展 `java.awt.Container`，所以 Swing 菜单（和菜单项）还是容器。

因为 Swing 菜单是按钮，所以它们继承了指定文本和一个图标的方法，而且因为 Swing 菜单还是容器，所以任何类型的组件都可以添加到一个菜单中。而 AWT 菜单只能显示文本。

AWT 菜单栏只可以添加到一个窗体中，即 AWT 菜单只能驻留在一个 Java 应用程序中。而 Swing 菜单栏可以添加到 Java 小应用程序和应用程序中。

表 10-4 列出了由 `java.awt.Menu` 实现的 public 方法和由 `JMenu` 实现的对应方法。

表 10-4 java.awt.Menu 方法和 JMenu 的对应方法

java.awt.Menu 方法	JMenu 的对应方法
<code>void add (MenuItem)</code>	<code>void add (JMenuItem)</code>
<code>void add (String)</code>	<code>void add (String)</code>
<code>void addSeparator ()</code>	<code>void addSeparator ()</code>
<code>MenuItem getItem (int)</code>	<code>JMenuItem getItem (int)</code>
<code>int getItemCount ()</code>	<code>int getItemCount ()</code>
<code>void insert (MenuItem, int)</code>	<code>JMenuItem insert (JMenuItem, int)</code>
<code>void insert (String, int)</code>	<code>void insert (String, int)</code>
<code>void insertSeparator (int)</code>	<code>void insertSeparator (int)</code>
<code>boolean isTearOff ()</code>	<code>boolean isTearOff ()</code>
<code>String paramString ()</code>	<code>String paramString ()</code>
<code>void remove (int)</code>	<code>void remove (int)</code>
<code>void remove (MenuComponent)</code>	<code>void remove (JMenuItem)</code>
<code>void removeAll ()</code>	<code>void removeAll ()</code>

注：对具有不同原型的方法用黑体字表示。

`JMenu` 维护与它的 AWT 对等组件（`java.awt.Menu`）的兼容。AWT 和 Swing 组件都可以添加（或插入）菜单项（AWT 菜单添加 AWT 菜单项，Swing 菜单添加 Swing 菜单项）和分隔线。而且，AWT 和 Swing 组件都提供对它们的菜单项数量的访问和对菜单项本身的访问。

激活 AWT 和 Swing 组件的菜单项时都激发动作事件。在选取或取消选取 Swing 菜单的菜单项时还激发菜单事件。

`JMenu` 和 `java.awt.Menu` API 都可移动菜单，但是在 Swing 1.1 FCS 中，没有为 Swing 菜单实现移动菜单功能。

10.7 菜单元素

JMenuItem、JMenu、JPopupMenu 和 JMenuBar 都实现 MenuElement 接口，以便参与菜单事件处理。虽然菜单和菜单项实现 MenuElement 接口，但是不需要理解这个接口，也不需要理解 JMenuItem 和 JMenu 是如何为每天使用的 Swing 菜单和菜单项实现这个接口的方法。

本节主要介绍使用 MenuElement 接口来实现一个定制的菜单元素。

接口总结 10-3 列出了 MenuElement 接口。

接口总结 10-3 MenuElement

```
public abstract Component getComponent ()
public abstract MenuElement [] getSubElements ()
public abstract void menuSelectionChanged (boolean)
public abstract void processKeyEvent (KeyEvent, MenuElement [], MenuSelectionManager)
public abstract void processMouseEvent (MouseEvent, MenuElement [], MenuSelectionManager)
```

MenuSelectionManager 的实例调用由 MenuElement 接口定义的所有方法，但不应该直接调用。MenuElement 接口的存在确保了菜单和菜单项参与了菜单事件的处理。

实现 MenuElement 接口

直接添加到菜单中的组件的行为与菜单项的行为不同；例如，在菜单的一个组件上拖动光标不会增亮组件。然而，实现 MenuElement 接口的组件可以参与菜单事件处理，甚至可以定义定制行为。图 10-17 所示的小应用程序图解说明实现定制菜单元素，即在元素文本上画下划线来指示这个元素已选取。

UnderlineElement 类扩展 JButton，并实现 MenuElement 接口。这个构造方法把这个元素的文本传送给 JButton 构造方法，并把它的边框设置为一个空边框。getComponent 方法返回与一个菜单元素相关联的组件，在 UnderlineElement 的例子中，这个组件就是组件本身。getSubElements 方法返回一个没有子项的 MenuElement 数组，指示 UnderlineElement 没有包含任何元素。

```
class UnderlineElement extends JButton implements MenuElement {
    private boolean drawUnderline = false;

    public UnderlineElement (String s) {
        super (s);
        setBorder (BorderFactory.createEmptyBorder (2, 2, 2, 2));
    }

    public Component getComponent () {
        return this;
    }

    public MenuElement [] getSubElements () {
        return new MenuElement [0];
    }

    ...
}
```

当改变菜单选取时，将调用 menuSelectionChanged 方法。传送给 menuSelectionChanged 的 boolean 变量代表这个元素是否被选取。

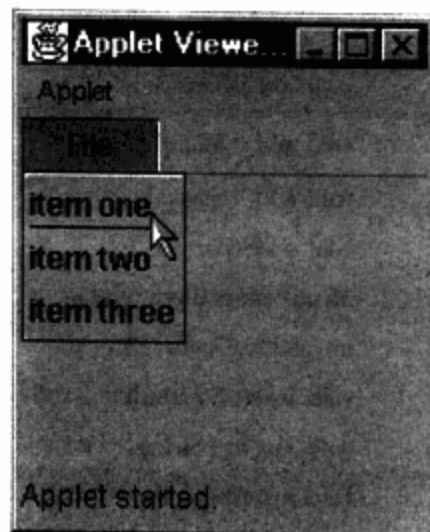


图 10-17 实现定制菜单元素

`UnderlineElement.menuSelectionChanged` 方法设置它的 `drawUnderline` 成员变量并调用 `repaint`。
`UnderlineElement` 还重载 `paintComponent`，以便在需要时在元素的文本上画下划线。

```
...
public void menuSelectionChanged (boolean b) {
    drawUnderline = b;
    repaint ();
}
public void paintComponent (Graphics g) {
    super.paintComponent (g);

    Insets insets = getInsets ();

    if (drawUnderline) {
        FontMetrics fm = g.getFontMetrics ();
        g.drawLine (insets.left, insets.top + fm.getHeight (),
                    fm.stringWidth (getText ()),
                    insets.top + fm.getHeight ());
    }
}
...
```

`UnderlineElement` 类对处理键事件不感兴趣，因此，在实现的 `processKeyEvent` 方法中没有写任何代码。

```
...
public void processKeyEvent (KeyEvent me,
                             MenuElement [] element,
                             MenuSelectionManager msm) {
}
...
```

`UnderlineElement` 重载 `processMouseEvent` 方法，以便调用 `super.processMouseEvent ()` 来确保这个元素的按钮状态被更新，然后调用 `MenuSelectionManager.processMouseEvent ()`。菜单选取管理器确定事件的接收者，检测在光标拖动时的进入/退出事件，并派发适当的事件。

上面所述都是需要的，因为当在一个菜单项上拖动鼠标时，菜单项会激发 `MenuDrag-MouseEvent`。当键按下、释放或输入时（当一个菜单项处于待命状态时），菜单项会激发 `MenuKeyEvents`。有关 `MenuDragMouseEvent` 的详细内容，请参见 10.6.4 节“JMenuitem 事件”。要记住的最重要的事是菜单元素必须重载 `processMouseEvent (MouseEvent)` 来调用 `MenuSelectionManager.processMouseEvent ()`。

```
...
public void processMouseEvent (MouseEvent me) {
    super.processMouseEvent (me);
    MenuSelectionManager defaultManager ().processMouseEvent (me);
}
...
```

根据事件类型，`UnderlineElement` 通过设置菜单选取管理器的选取路径来实现由 `MenuElement` 接口定义的 `processMouseEvent` 方法。

如果这个事件是鼠标点击或释放事件，则路径设置为 `null` 并且调用 `doClick ()`，并激活这个按钮。有关 `doClick` 方法的详细内容，请参见“类总结 8-4 Abstract Button”。

如果事件是其他类型的鼠标事件，例如，鼠标进入或退出事件，则以元素为参数来构造路径。这个路径必须包括元素本身，以便这个元素能接收以后来自菜单选取管理器的事件。

```

...
public void processMouseEvent (MouseEvent me,
                               MenuElement [] element,
                               MenuSelectionManager msm) {
    if (me.getID () == MouseEvent.MOUSE_CLICKED ||
        me.getID () == MouseEvent.MOUSE_RELEASED) {
        msm.setSelectedPath (null);
        doClick ();
    }
    else
        msm.setSelectedPath (getPath ());
}

public MenuElement [] getPath () {
    MenuSelectionManager defaultManager =
        MenuSelectionManager defaultManager ();
    MenuElement oldPath [] = defaultManager.getSelectedPath ();
    MenuElement newPath [];
    int len = oldPath.length;
    if (len > 0) {
        MenuElement lastElement = oldPath [len-1];
        Component parent = getParent ();
        if (lastElement == parent) {
            newPath = new MenuElement [len + 1];
            System.arraycopy (oldPath, 0, newPath, 0, len);
            newPath [len] = this;
        }
        else {
            int j;
            for (j = len-1; j >= 0; j--) {
                if (oldPath [j].getComponent () == parent)
                    break;
            }
            newPath = new MenuElement [j+2];
            System.arraycopy (oldPath, 0, newPath, 0, j+1);
            newPath [j+1] = this;
        }
    }
    else
        return new MenuElement [0];
    return newPath;
}

```

实现定制菜单元素看起来相当复杂，但是这个过程实际上就是做两件事。

第一，菜单元素必须重载 `processMouseEvent (MouseEvent)` 来调用 `MenuSelectionManager.processMouseEvent ()`，以便菜单选取管理器可以执行适当的内部管理和事件激发。

第二，菜单元素应该重载 `processMouseEvent (MouseEvent, MenuElement [], MenuSelectionManager)` 来实际处理事件。如果菜单元素想要接收以后来自菜单选取管理器的事件，则它必须把自己添加到传送给菜单元素的路径中。

例 10-15 列出了图 10-17 所示的小应用程序的完整代码。

例 10-15 实现定制菜单元素

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JMenuBar menuBar = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");

        fileMenu.add (new UnderlineElement ("item one"));
        fileMenu.add (new UnderlineElement ("item two"));
        fileMenu.add (new UnderlineElement ("item three"));

        menuBar.add (fileMenu);
        setJMenuBar (menuBar);
    }
}

class UnderlineElement extends JButton implements MenuElement {
    private boolean drawUnderline = false;

    public UnderlineElement (String s) {
        super (s);
        setBorder (BorderFactory.createEmptyBorder (2, 2, 2, 2));
    }

    public Component getComponent () {
        return this;
    }

    public MenuElement [] getSubElements () {
        return new MenuElement [0];
    }

    public void menuSelectionChanged (boolean b) {
        drawUnderline = b;
        repaint ();
    }

    public void paintComponent (Graphics g) {
        super.paintComponent (g);

        Insets insets = getInsets ();

        if (drawUnderline) {
            FontMetrics fm = g.getFontMetrics ();
            g.drawLine (insets.left, insets.top + fm.getHeight (),
                        fm.stringWidth (getText ()),
                        insets.top + fm.getHeight ());
        }
    }

    public void processKeyEvent (KeyEvent me,
                                MenuElement [] element,
                                MenuSelectionManager msm) {
    }

    public void processMouseEvent (MouseEvent me) {
        super.processMouseEvent (me);
        MenuSelectionManager defaultManager ().processMouseEvent (me);
    }
}

```

```

    }
    public void processMouseEvent (MouseEvent me,
                                   MenuElement [] element,
                                   MenuSelectionManager msm) {
        if (me.getID () == MouseEvent.MOUSE_CLICKED ||
            me.getID () == MouseEvent.MOUSE_RELEASED) {
            msm.setSelectedPath (null);
            doClick ();
        }
        else
            msm.setSelectedPath (getPath ());
    }
    public MenuElement [] getPath () {
        MenuSelectionManager defaultManager =
            MenuSelectionManager.defaultManager ();
        MenuElement oldPath [] = defaultManager.getSelectedPath ();
        MenuElement newPath [];
        int len = oldPath.length;
        if (len > 0) {
            MenuElement lastElement = oldPath [len-1];
            Component parent = getParent ();
            if (lastElement == parent) {
                newPath = new MenuElement [len+1];
                System.arraycopy (oldPath, 0, newPath, 0, len);
                newPath [len] = this;
            }
            else {
                int j;
                for (j = len-1; j >= 0; j--) {
                    if (oldPath [j].getComponent () == parent)
                        break;
                }
                newPath = new MenuElement [j+2];
                System.arraycopy (oldPath, 0, newPath, 0, j+1);
                newPath [j+1] = this;
            }
        }
        else
            return new MenuElement [0];
        return newPath;
    }
}

```

10.8 JPopupMenu

我们知道菜单是按钮，当鼠标指针进入菜单时，它显示一个弹出式菜单。弹出式菜单（由 JPopupMenu 类代表）除用于菜单外，还有别的用途，即一个弹出式菜单可以在一个组件内的任何地方显示，也可以在相对于屏幕的任何地方显示。

可以把动作、菜单项、组件和分隔线添加到一个 Swing 弹出式菜单中。当把一个动作添加

到一个弹出式菜单中时，这个弹出式菜单将创建一个菜单项，并把它添加到这个弹出式菜单中。有关把动作添加到 Swing 组件中的详细内容，请参见 5.3 节“动作”。

图 10-18 所示的小应用程序显示包含两个菜单项、一个分隔线和一个组件的弹出式菜单。

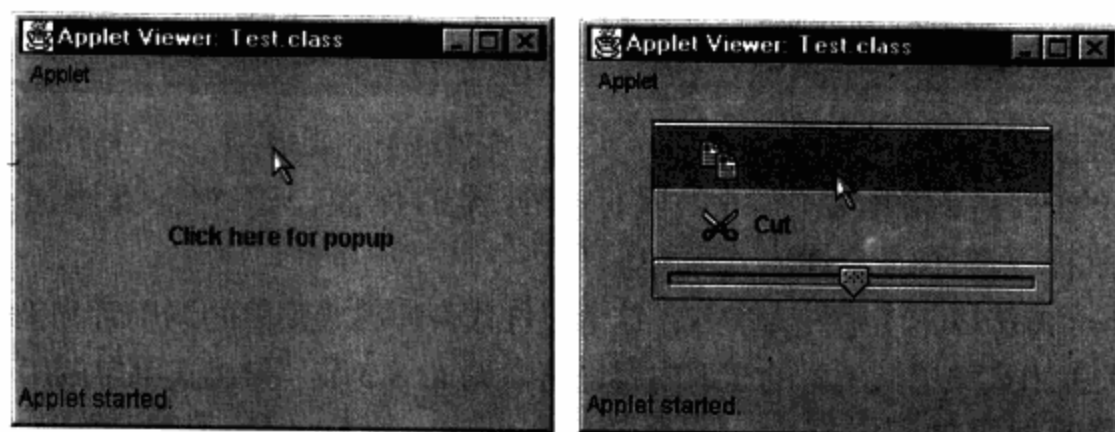


图 10-18 带两个菜单项、一个分隔线和一个组件的弹出式菜单

例 10-16 列出了图 10-18 所示的小应用程序的代码。

例 10-16 使用弹出式菜单

```
import javax.swing.*.*;
import javax.swing.event.*;
import java.awt.*.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        final JLabel label = new JLabel ("Click here for popup");
        final JPopupMenu popup = new JPopupMenu ();
        final JSlider slider = new JSlider ();

        popup.add (new JMenuItem ("Copy",
                                   new ImageIcon ("copy.gif")));
        popup.add (new CutAction ());
        popup.addSeparator ();
        popup.add (slider);

        label.addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                popup.show (label, e.getX (), e.getY ());
            }
        });

        slider.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                if (! slider.getValueIsAdjusting ())
                    popup.setVisible (false);
            }
        });

        label.setHorizontalAlignment (JLabel.CENTER);
        contentPane.add (label, BorderLayout.CENTER);
    }

    class CutAction extends AbstractAction {
        public CutAction () {
            super ("Cut", new ImageIcon ("cut.gif"));
        }
    }
}
```

```

    |
    |
    | public void actionPerformed (ActionEvent e) |
    |     System.out.println ("cut");
    |
    |
    |

```

这个小应用程序创建了一个 JPopupMenu 实例，它带两个菜单项、一个分隔线和一个滑杆。这个小应用程序的内容窗格配备了一个鼠标监听器的标签，这个监听器通过调用 JPopupMenu.show() 在鼠标下显示这个弹出式菜单。

JPopupMenu.show 方法以一个组件为参数，这个组件代表弹出式菜单的调用者和坐标（即相对于调用者的坐标）。有关弹出式菜单调用者的详细内容，请参见 10.8.3 节“弹出式菜单调用者”。

把一个变化监听器添加到滑杆中，这个滑杆在调整了滑杆的值后隐藏弹出式菜单。有关滑杆的详细内容，请参见第 11 章“进度条、滑杆和分隔条”。

“剪切”和“拷贝”菜单项都显示一个图标和一个字符串。通过把一个动作添加到弹出式菜单中来创建“剪切”菜单项，通过把一个 JMenuItem 实例进行实例化来创建“拷贝”菜单项。这两种方式创建了几乎完全相同的菜单项，这说明了菜单项和动作都可以添加到一个弹出式菜单中。

10.8.1 弹出式菜单触发器

虽然有时需要显示一个弹出式菜单以响应鼠标按下事件，如例 10-16 所列的小应用程序那样，但是，通常显示弹出式菜单是为了响应称作弹出式菜单触发器的事件序列。弹出式菜单触发器与窗口系统有关；例如，下面介绍了 Motif 和 Windows 的弹出式菜单触发器：

- **Motif 弹出式菜单触发器：**当鼠标按钮 3 按下时，如果这个鼠标按钮被按住或在短时间内释放的话，显示弹出式菜单并保持显示状态。此后，在弹出式菜单外或在弹出式菜单的一个菜单项内再按下鼠标使弹出式菜单消失。

- **Windows 弹出式菜单触发器：**在鼠标按钮 2 按下时显示弹出式菜单。此后，在弹出式菜单外或在弹出式菜单的一个菜单项内再单击鼠标按钮 1 或鼠标按钮 2 使弹出式菜单消失。

例 10-17 所列的小应用程序列出了响应弹出式菜单触发器的一个弹出式菜单。

例 10-17 显示一个弹出式菜单以响应弹出式菜单触发器

```

import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {
    private JPopupMenu popup = new JPopupMenu ();

    public void init () {
        Container contentPane = getContentPane ();

        popup.add (new JMenuItem ("item one"));
        popup.add (new JMenuItem ("item two"));
        popup.add (new JMenuItem ("item three"));
        popup.add (new JMenuItem ("item four"));

        contentPane.addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                showPopup (e);
            }
        });
    }
}

```



```

        public void mouseClicked (MouseEvent e) {
            showPopup (e);
        }
        public void mouseReleased (MouseEvent e) {
            showPopup (e);
        }
    };

    void showPopup (MouseEvent e) {
        if (e.isPopupTrigger ())
            popup.show (this, e.getX (), e.getY ());
    }
}

```

这个小应用程序实现一个鼠标监听器，它处理鼠标按下、点击和释放事件。通过调用这个小应用程序的 `showPopup` 方法来处理每个事件，`showPopup` 方法又调用 `MouseEvent.isPopupTrigger` 方法来检测弹出式触发器。如果触发器被“触发”，则显示弹出式菜单。这个小应用程序重载了 `java.awt.event.MouseListener` 定义的三个处理鼠标按下和点击的方法，以检查弹出式菜单的触发器。重载这三个方法是必须的，因为弹出式菜单的触发器可以关联不同的鼠标事件。

注意 检测弹出式菜单触发器有些难。更好的解决方法可能是实现一个弹出式菜单的触发器事件。这样，可以向一个组件登记一个弹出式菜单触发监听器，而且不必处理一般的鼠标事件。这种方法不仅更方便，而且更符合基于委托的事件模型。

Swing 提示

使用弹出式菜单触发器来显示弹出式菜单

窗口系统定义一个弹出式菜单触发器（一个事件序列，它显示一个弹出式菜单）。当显示一个弹出式菜单时，虽然不需要检测弹出式菜单的触发器，但是通常希望弹出式菜单只在响应弹出式菜单触发器时才显示。因此，好的 GUI 设计表明弹出式菜单（由菜单显示的弹出式菜单除外）应当只在响应弹出式菜单触发器时才显示。

10.8.2 轻量/中量/重量弹出式菜单

对图 10-18 所示的弹出式菜单采用了定位策略，以便使这个弹出式菜单完全包含在小应用

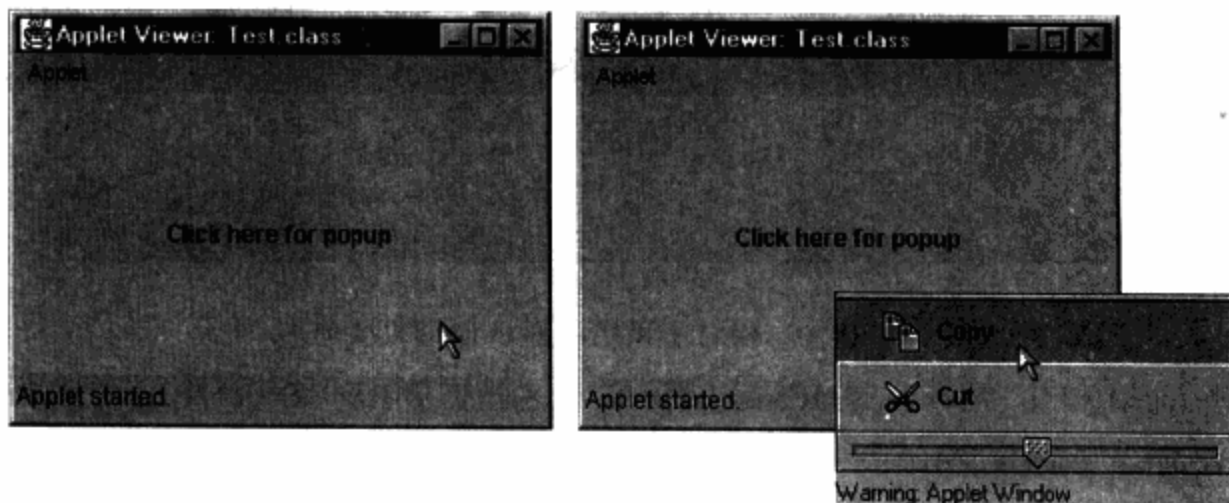


图 10-19 一个重量弹出式菜单

程序的窗口中。如果在定位这个弹出式菜单时，其边界超出了所在窗口的边界，则这个弹出式菜单将包含在它自己的一个窗口中，如图 10-19 所示。

Swing 弹出式菜单只用一个类（JPopupMenu）来表示，但实际上有三种不同的弹出式菜单，如表 10-5 所示。

有三种不同类型的 Swing 弹出式菜单的原因如下：

表 10-5 Swing 弹出式菜单类型

弹出式菜单类型	把弹出式菜单显示在 ... 中
轻量	一个轻量容器
中量	一个重量 AWT 面板
重量	一个 JWindow 实例

首先，如果轻量 Swing 弹出式菜单与重量 AWT 组件重叠，则轻量弹出式菜单将在 AWT 组件下面显示。这种显示方式与层序（分配给组件的深度）有关，即重量组件的层序比轻量组件的层序高。如果要在重量组件上显示弹出式菜单，则显示弹出式菜单的容器必须是重量的 AWT 容器。有关层序的解释及混合使用轻量组件和重量组件的缺点的介绍，请参见 2.3 节“混合使用 Swing 和 AWT 组件”。

如果一个弹出式菜单没有完全包含在它的上层窗口中，则轻量和中量弹出式菜单将在窗口的边界上显示。结果，没有完全在窗口中的弹出式菜单将在它们自己的本地窗口中显示。

因此，当显示一个 Swing 弹出式菜单时，下面的算法确定了显示弹出式菜单的容器的类型：

如果弹出式菜单没有完全显示在它的上层窗口中，则使用一个重量弹出式菜单，即弹出式菜单将显示在它自己的窗口中以确保在显示弹出式菜单时，整个菜单都是可见的。

如果弹出式菜单完全在它的上层窗口中，则根据弹出式菜单的轻量弹出式菜单的允许属性来决定是使用轻量弹出式菜单还是中量弹出式菜单。

10.8.3 弹出式菜单调用者

弹出式菜单有一个与之相关联的组件，这个组件称作弹出式菜单的调用者。弹出式菜单的显示位置通常是相对于它们的调用者的，而且为了许多其他的事情，JPopupMenu 内部还使用调用者，例如，用调用者来定位弹出式菜单所在的窗口，以确定是否需要把弹出式菜单包含在它自己的一个窗口中。

图 10-20 所示的小应用程序显示一个弹出式菜单。当在组合框中选取了一个选项时，则在组合框所选取的画布中显示弹出式菜单。

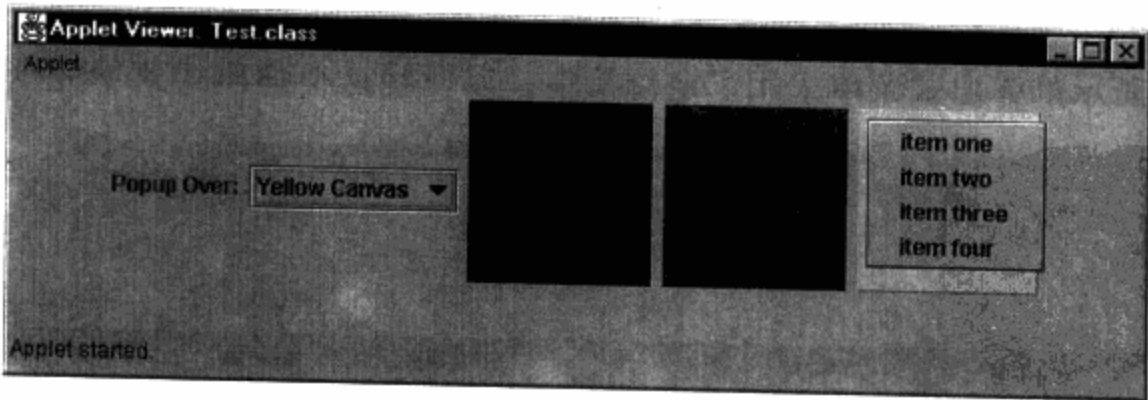


图 10-20 相对于调用者显示弹出式菜单

其中那些带颜色的“画布”是 JPanel 的扩展，这些扩展用一个 3D 矩形来填充它们的内部。

```
class ColoredCanvas extends JPanel {
    private Color color;

    public ColoredCanvas (Color color) {
```

```

        this.color = color;
    }
    public void paintComponent (Graphics g) {
        super.paintComponent (g);

        Dimension size = getSize ();
        g.setColor (color);
        g.fillRect (0, 0, size.width-1, size.height-1, true);
    }
    public Dimension getPreferredSize () {
        return new Dimension (100, 100);
    }
}

```

把一个子项监听器添加到这个组合框中，这个监听器把弹出式菜单显示在相对于画布的位置上，这块画布是从组合框中选取的画布。

```

comboBox.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        JComboBox c = (JComboBox) event.getSource ();
        String label = (String) c.getSelectedItem ();

        if (label.equals ("Blue Canvas"))
            popupRelativeToMe = blueCanvas;
        else if (label.equals ("Red Canvas"))
            popupRelativeToMe = redCanvas;
        else if (label.equals ("Yellow Canvas"))
            popupRelativeToMe = yellowCanvas;

        popup.show (popupRelativeToMe, 5, 5);
    }
});

```

例 10-18 列出了图 10-20 所示的小应用程序的完整代码。

例 10-18 相对于其调用者显示弹出式菜单

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    JComboBox comboBox = new JComboBox ();
    JPopupMenu popup = new JPopupMenu ();
    ColoredCanvas popupRelativeToMe;
    ColoredCanvas blueCanvas, redCanvas, yellowCanvas;

    public void init () {
        Container contentPane = getContentPane ();
        blueCanvas = new ColoredCanvas (Color.blue);
        redCanvas = new ColoredCanvas (Color.red);
        yellowCanvas = new ColoredCanvas (Color.yellow);
        popupRelativeToMe = blueCanvas;

        popup.add (new JMenuItem ("item one"));
        popup.add (new JMenuItem ("item two"));
        popup.add (new JMenuItem ("item three"));
        popup.add (new JMenuItem ("item four"));

        contentPane.setLayout (new FlowLayout ());
    }
}

```

```

contentPane.add (new JLabel ("Popup Over:"));
contentPane.add (combobox);
contentPane.add (blueCanvas);
contentPane.add (redCanvas);
contentPane.add (yellowCanvas);

combobox.addItem ("Blue Canvas");
combobox.addItem ("Yellow Canvas");
combobox.addItem ("Red Canvas");

combobox.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        if (event.getStateChange () == ItemEvent.SELECTED) {
            JComboBox c = (JComboBox) event.getSource ();
            String label = (String) c.getSelectedItem ();

            if (label.equals ("Blue Canvas"))
                popupRelativeToMe = blueCanvas;
            else if (label.equals ("Red Canvas"))
                popupRelativeToMe = redCanvas;
            else if (label.equals ("Yellow Canvas"))
                popupRelativeToMe = yellowCanvas;

            popup.show (popupRelativeToMe, 5, 5);
        }
    }
});
}

class ColoredCanvas extends JPanel {
    private Color color;

    public ColoredCanvas (Color color) {
        this.color = color;
    }

    public void paintComponent (Graphics g) {
        super.paintComponent (g);

        Dimension size = getSize ();
        g.setColor (color);
        g.fillRect (0, 0, size.width-1, size.height-1, true);
    }

    public Dimension getPreferredSize () {
        return new Dimension (100, 100);
    }
}

```

组件总结 10-5 总结了 JPopupMenu 类。

组件总结 10-5 JPopupMenu

模型: SingleSelectionModel
 UI 代表: javax.swing.plaf.basic.BasicPopupMenuUI
 绘制器: ——
 编辑器: ——
 激发的事件: PropertyChangeEvent/PopupMenuEvent

替换: java.awt.PopupMenu

类图:

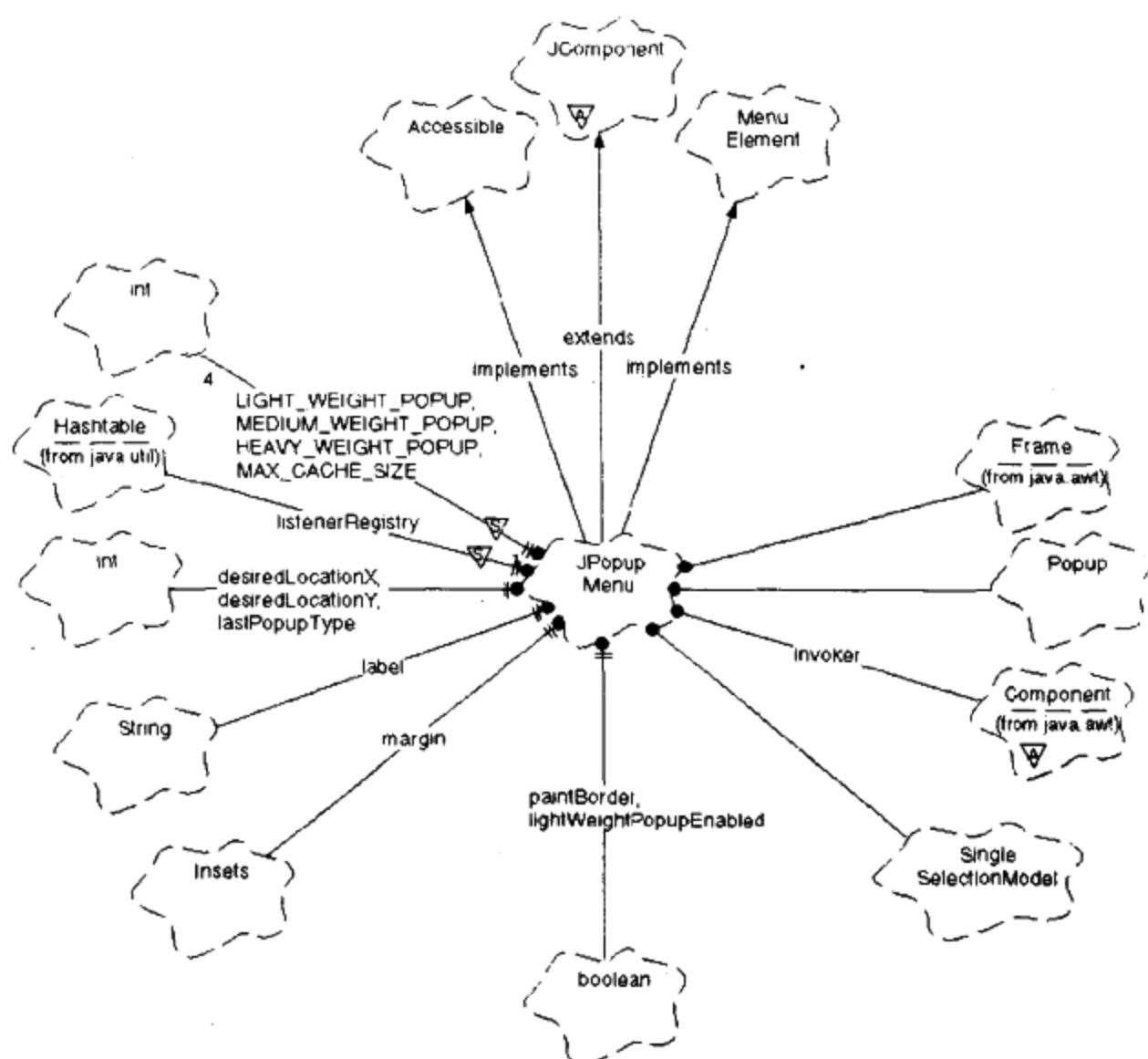


图 10-21 JPopupMenu 类图

JPopupMenu 扩展 JComponent，并实现 Accessible 接口和 MenuElement 接口。JPopupMenu 维护对一个 Insets 实例的 private 引用，这个实例代表弹出式菜单的边框和显示的菜单项之间的边衬。JPopupMenu 还维护对一个字符串的 private 引用和对一些 boolean 属性的 private 引用，其中把字符串用作弹出式菜单的标题，boolean 属性用来确定 JPopupMenu 是否绘制它的边框以及这个弹出式菜单是否显示在一个轻量容器中。是否使用弹出式菜单的标题取决于弹出式菜单的界面样式。对 Swing 1.1 FCS 而言，标准的 Swing 界面样式都不显示标题。

JPopupMenu 跟踪它的调用者，跟踪显示弹出式菜单的组件，并跟踪调用者的窗体。缺省时，弹出式菜单的调用者是 null（即弹出式菜单没有被包含在组件中），但是弹出式菜单的调用者可以显式地设置，如例 10-18 所示。JPopupMenu 实例的模型是一个 SingleSelectionModel，它跟踪当前选取了哪一个菜单项。

JPopupMenu 还维护对 Popup 的一个引用，这个引用是在 JPopupMenu 类中定义的一个 private 接口。还使用了 Popup 接口，以便 JPopupMenu 可以操纵实际的弹出式菜单，不管它是重量、中量或轻量弹出式菜单，都能够以统一的方式来操纵。因为 Popup 接口是 JPopupMenu 类中的 private 接口，所以在此就不介绍了。

10.8.4 JPopupMenu 属性

表 10-6 列出了由 JPopupMenu 类维护的属性。

表 10-6 JPopupMenu 属性

属性名	数据类型	访问 ^①	类型 ^②	缺省 ^③
borderPainted	int	SG	S	true
componentAtIndex	Component	G	S	—
componentIndex	int	G	I	—
invoker	Component	SG	S	null
label	String	CSG	B	null
lightWeightPopupEnabled	boolean	SG	S	true
margin	Insets	G	S	(0, 0, 0, 0)
popupSize	Dimension	S	S	L&F
rootPopupMenu	JPopupMenu	G	S	—
selectionModel	SingleSelectionModel	SG	S	DefaultSingle- SelectionModel

① C = 可在构造时刻设置/G = 获取方法/S = 设置方法
② B = 关联的（激发 PropertyChangeEvent）/C = 受约束的/I = 索引的/S = 简单的
③ L&F = 与界面样式有关

borderPainted——确定是否绘制弹出式菜单的边框。

componentAtIndex、componentIndex——通过 componentAtIndex 和 componentIndex 属性，弹出式菜单可以分别提供已知索引值的组件和已知组件的索引值。两种设置方法都使用 JPopupMenu 从 java.awt.Container 继承的 getComponents 方法来获得组件数组，这些组件包含在弹出式菜单中。

如果 componentAtIndex 以一个无效的索引为参数，则从这个方法中返回一个 null 引用。如果 getComponentIndex 以一个不包含在弹出式菜单中的组件为参数，则这个方法返回-1。

invoker——通常通过使用 JPopupMenu.show (Component invoker, int x, int y) 把弹出式菜单显示在相对于它们的调用者组件的坐标上，show 方法在调用者坐标系统中显示弹出式菜单。

使用 JPopupMenu.setLocation (int x, int y) 方法，还可以把弹出式菜单显示在相对于屏幕的坐标上，其中的 x、y 是屏幕坐标。调用 JPopupMenu.setVisible () 而不是 JPopupMenu.show () 也可以使弹出式菜单显示在相对于屏幕的坐标上。然而，所有的弹出式菜单必须有一个非 null 的调用者，因此，用 setLocation () 定位和用 setVisible () 显示的弹出式菜单必须用 JPopupMenu.setInvoker 方法来显式地指定一个调用者。

label——弹出式菜单的标签用作菜单的标题。显示弹出式菜单的标题由弹出式菜单的界面样式负责。在 Swing 1.2 FCS 中，界面样式都不显示弹出式菜单标题。

lightWeightPopupEnabled——完全包含在一个窗口中的弹出式菜单可以在轻量容器或重量容器中显示。如果一个弹出式菜单的 lightWeightPopupEnabled 属性是 true，则使用轻量容器，如果这个属性是 false，则这个弹出式菜单将包含在一个 AWT 面板中。

margin——margin 属性代表从弹出式菜单的内容的内边沿到包含弹出式菜单的组件的外边框之间的边衬。

popupSize——JPopupMenu 类提供了一个设置方法来设置弹出式菜单的大小，但没有提供获取方法。

rootPopupMenu——rootPopupMenu 属性是一个计算属性，它返回 JPopupMenu 的最顶层父组件。

selectionModel——JPopupMenu 类的选取模型是 SingleSelectionModel 类型。缺省时，弹出式

菜单配备了一个 DefaultSingleSelectionModel 实例。

因为弹出式菜单是短暂的，即它们不占用容器中永久的资源，所以设置由 JPopupMenu 维护的大多数属性不会更新弹出式菜单的显示。例如，设置选取属性不会使弹出式菜单重新绘制。这与其他可选取的 Swing 组件不同，为可选取的 Swing 组件设置选取状态通常会使组件的显示更新。

10.8.5 JPopupMenu 事件

JPopupMenu 的实例激发两种类型的事件：PropertyChangeEvents 和 PopupMenuEvents。当与弹出式菜单相关联的关联属性修改时（与所有 Swing 组件的情况相同），将激发 PropertyChangeEvents。例如，当一个弹出式菜单的可见性修改时，这个弹出式菜单将激发一个属性变化事件。

当弹出式菜单将变成可见或不可见时，或当弹出式菜单取消时，将激发 PopupMenuEvents。在弹出式菜单可见的时，接着点击这个菜单外面的区域会使这个菜单不可见（即没有从弹出式菜单中选取菜单项），弹出式菜单就取消了。

Swing.event 包定义一个 PopupMenuListener 接口。接口总结 10-4 列出了由 PopupMenuListener 接口定义的方法。

接口总结 10-4 PopupMenuListener

扩展：java.util.EventListener

```
public abstract void popupMenuCanceled (PopupMenuEvent)
public abstract void popupMenuWillBecomeInvisible (PopupMenuEvent)
public abstract void popupMenuWillBecomeVisible (PopupMenuEvent)
```

当弹出式菜单已经取消时，将调用由 PopupMenuEvent 类定义的这些方法。

图 10-22 所示的小应用程序把一个弹出式菜单监听器添加到一个 JPopupMenu 实例中。当弹出式菜单成为可见的或不可见的或取消时，这个监听器显示一个消息。图 10-22 中左图显示在弹出式菜单已经可见后小应用程序的样子。右图显示在弹出式菜单已经不可见后小应用程序的样子。例 10-19 列出了图 10-22 所示的小应用程序的代码。

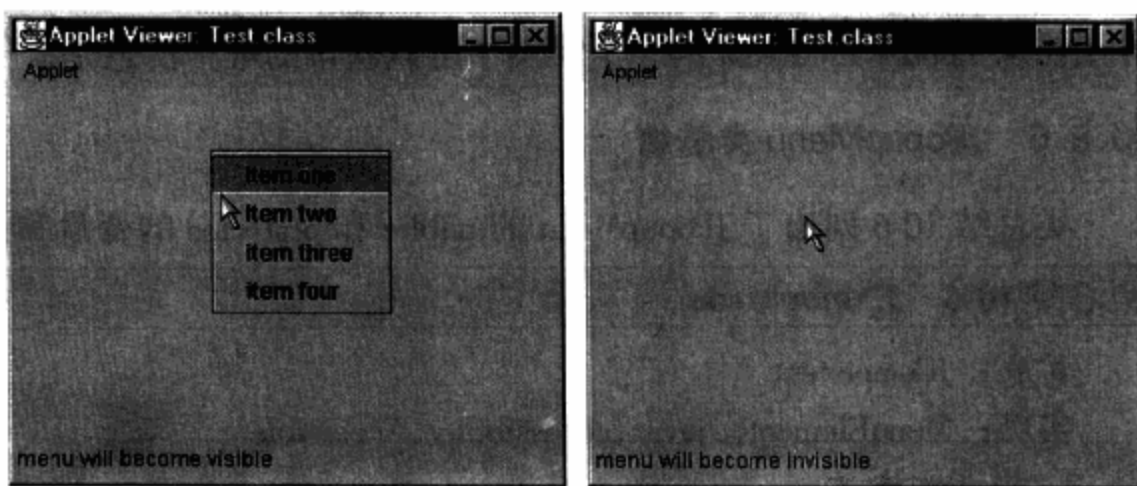


图 10-22 弹出式菜单事件

例 10-19 弹出式菜单事件的清单

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
```



```

public class Test extends JApplet {
    public void init () {
        final Container contentPane = getContentPane ();
        final JPopupMenu popup = new JPopupMenu ();

        popup.add (new JMenuItem ("item one"));
        popup.add (new JMenuItem ("item two"));
        popup.add (new JMenuItem ("item three"));
        popup.add (new JMenuItem ("item four"));

        popup.addPopupMenuListener (new PopupMenuListener () {
            public void popupMenuCanceled (PopupMenuEvent e) {
                System.out.println ("menu canceled");
            }
            public void popupMenuWillBecomeVisible (
                PopupMenuEvent e) {
                System.out.println ("menu will become visible");
            }
            public void popupMenuWillBecomeInvisible (
                PopupMenuEvent e) {
                System.out.println ("menu will become invisible");
            }
        });
        addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                popup.show (contentPane, e.getX (), e.getY ());
            }
        });
    }
}

```

与 MenuEvent 类一样, PopupMenuEvent (它扩展 java.util.EventObject 类) 没有提供它自己的方法。从 PopupMenuEvent 的一个实例中可以获得的唯一信息是事件源。

10.8.6 JPopupMenu 类总结

类总结 10-6 列出了 JPopupMenu 的 public 和 protected 的变量和方法。

类总结 10-6 JPopupMenu

扩展: JComponent

实现: MenuElement、javax.accessibility.Accessible

1. 构造方法

```

public JPopupMenu ()
public JPopupMenu (String)

```

JPopupMenu 类提供了两个构造方法, 一个是无参数的, 另一个以代表菜单标题的字符串为参数。根据与弹出式菜单有关的界面样式来决定是否实际使用指定为弹出式菜单的标题的字符串。在 Swing 1.1 FCS 中, 标准的界面样式都不显示弹出式菜单的标题。

2. 方法

(1) 轻量弹出式菜单

```

public static boolean getDefaultLightWeightPopupEnabled ()
public static void setDefaultLightWeightPopupEnabled (boolean)
public void setLightWeightPopupEnabled (boolean)

```

上面所列的这些方法控制显示弹出式菜单的容器类型。Static `setDefaultLightWeightPopupEnabled` 方法为所有在调用这个方法后创建的弹出式菜单设置容器类型。如果这个方法以 `true` 为参数, 则此后创建的弹出式菜单将是轻量的弹出式菜单 (如果这个菜单完全在它的上级窗口中)。如果这个方法以 `false` 为参数, 则此后创建的、完全在它的上级窗口中的弹出式菜单将被包含在一个 AWT 重量面板中。

可以为每个弹出式菜单调用 `setLightWeightPopupEnabled` 方法, 以便在需要时重载通过 `setDefaultLightWeightPopupEnabled` 设置的缺省行为。例如, 在调用 `setDefaultLightWeightPopupEnabled (true)` 后创建的所有弹出式菜单都将是轻量的, 然而, 如果为一个特定的弹出式菜单实例调用 `setLightWeightPopupEnabled (false)`, 则这个弹出式菜单将被包含在一个 AWT 重量面板中 (或被包含在 `JFrame` 的一个实例中, 如果这个弹出式菜单没有完全在它的上级窗口中的话)。

有关确定显示弹出式菜单的容器的类型的算法更简洁的描述, 请参见 10.8 节 “JPopupMenu”。

(2) 添加和插入对象

```
public JMenuItem add (Action)
public JMenuItem add (JMenuItem)
public Component add (Component)
public void addSeparator ()
public void insert (Action, int index)
public void insert (Component, int index)
```

上面所列的这些方法被用于在一个弹出式菜单中添加和插入对象。可以把动作、菜单项、组件和分隔线添加到一个弹出式菜单中, 动作和组件可以插入在一个指定的索引中。

当把一个动作添加到一个弹出式菜单中时, 使用这个动作名和图标来创建一个后续要添加到这个弹出式菜单中的一个菜单项。

(3) 监听器/事件激发

```
public void addPopupMenuListener (PopupMenuListener)
public void removePopupMenuListener (PopupMenuListener)
protected PropertyChangeListener createActionChangeListener (JMenuItem)
protected void firePopupMenuCanceled ()
protected void firePopupMenuWillBecomeInvisible ()
protected void firePopupMenuWillBecomeVisible ()
```

弹出式菜单激发 `PopupMenuEvent` 事件, 这些事件被 `PopupMenuListener` 的实例所监听。有关被弹出式菜单激发的事件的详细内容, 请参见本章中 10.8.5 节 “JPopupMenu 事件”。

(4) 属性访问方法

```
public Component getComponent ()
public Component getComponentAtIndex (int)
public int getComponentIndex (Component)
public Component getInvoker ()
public String getLabel ()
public Insets getMargin ()
public SingleSelectionModel getSelectionModel ()
public MenuElement [] getSubElements ()
public void setBorderPainted (boolean)
public void setInvoker (Component)
public void setLabel (String)
public void setLocation (int, int)
public void setPopupSize (int, int)
```

```

public void setPopupSize (Dimension)
public void setSelected (Component)
public void setSelectionModel (SingleSelectionModel)
public void setVisible (boolean)

public boolean isBorderPainted ()
public boolean isLightWeightPopupEnabled ()
public boolean isVisible ()

```

上面所列的这些方法是 JPopupMenu 属性的访问方法，JPopupMenu 属性的详细内容在 10.8.4 节“JPopupMenu 属性”中介绍。

图 10-23 所示的小应用程序为它显示的弹出式菜单设置下面的属性：是否绘制弹出式菜单的边框、调用弹出式菜单的组件、和弹出式菜单的位置。

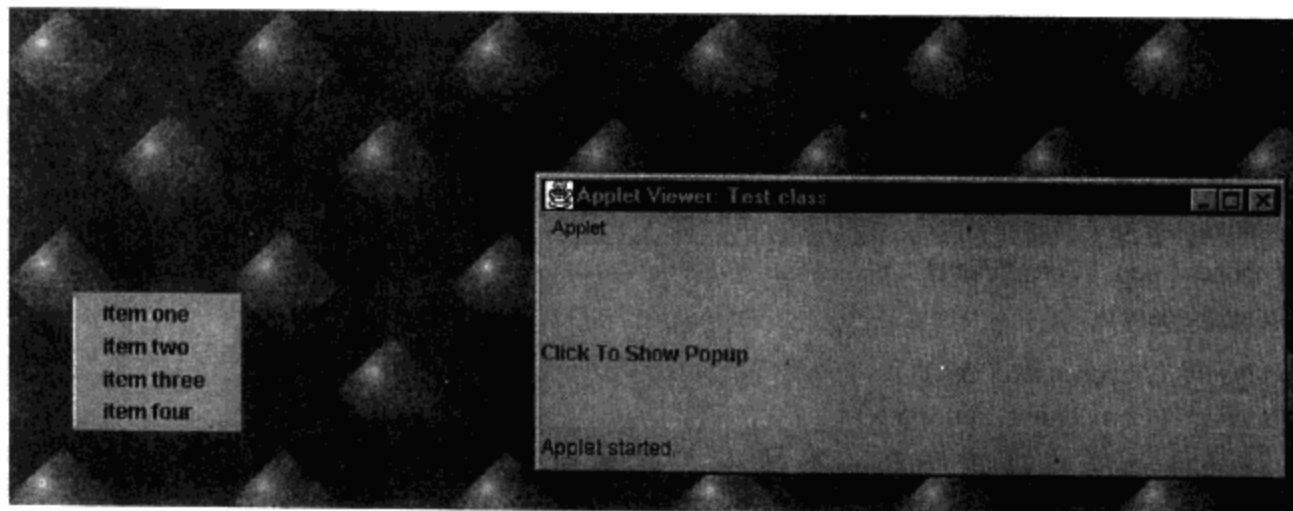


图 10-23 用屏幕坐标来指定一个弹出式菜单的位置

例 10-20 列出了图 10-23 所示的小应用程序的代码。

例 10-20 为弹出式菜单设置位置、边框和调用者

```

import javax.swing. * ;
import java.awt. * ;
import java.awt.event. * ;

public class Test extends JApplet {
    private JPopupMenu popup = new JPopupMenu ();

    public void init () {
        Container contentPane = getContentPane ();
        JLabel label = new JLabel ("Click To Show Popup");
        Listener listener = new Listener (popup, label);

        popup.add (new JMenuItem ("item one"));
        popup.add (new JMenuItem ("item two"));
        popup.add (new JMenuItem ("item three"));
        popup.add (new JMenuItem ("item four"));

        contentPane.add (label);
        label.addMouseListener (listener)
    }

    class Listener extends MouseAdapter {
        private JPopupMenu popup;
        private Component component;

        public Listener (JPopupMenu popup, Component component) {
            this.popup = popup;

```

```

        this.Component = component;

        popup.setBorderPainted (false);
        popup.setInvoker (component);
        popup.setLocation (200, 200);
    }

    public void mousePressed (MouseEvent e) {
        popup.setVisible (true);
    }
}

```

这个小应用程序实现了一个监听器，它扩展 `MouseAdapter` 类（来自 `java.awt.event` 包）。把这个监听器添加到在小应用程序中间显示的标签中。当在标签中按下鼠标时，弹出式菜单的可见性设置为 `true`。当构造监听器时，它以对这个弹出式菜单的一个引用为参数，边框绘制设置为 `false`，调用者设置为这个标签，弹出式菜单的位置设置为 `(200, 200)`。

注意 为弹出式菜单指定的位置是屏幕坐标，不是相对于调用者的坐标。

(5) `MenuItem` 接口

```

public Component getComponent ()
public MenuItem [] getSubElements ()
public void menuSelectionChanged (boolean)
public void processKeyEvent (KeyEvent, MenuItem [], MenuSelectionManager)
public void processMouseEvent (MouseEvent, MenuItem [], MenuSelectionManager)

```

上面所列的这些方法在 `MenuItem` 接口中定义。`processKeyEvent` 和 `processMouseEvent` 方法由 `JPopupMenu` 以无操作形式实现。

(6) 显示/包装/绘制边框

```

public void show (Component invoker, int x, int y)
public void pack ()
protected void paintBorder (Graphics)

```

上面所列的 `show` 方法相对于传送给它的组件显示一个弹出式菜单。传送给 `show` 方法的两个 `integer` 值指定弹出式菜单的显示位置（相对于这个组件的左上角）。

`pack` 类似于 `java.awt.Window.pack()`（有关包装窗口的详细内容，请参见 14.1 节“`JWindow`”），且可把弹出式菜单调整为它的首选大小。从 `JComponent` 中重载 `paintBorder` 方法，只有绘制属性设置为 `true` 才会绘制弹出式菜单的边框。

(7) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()
public PopupMenuUI getUI ()
public void setUI (PopupMenuUI)

```

上面列出的方法可以在大多数 `JComponent` 扩展中找到。`Swing` 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。`updateUI` 方法在组件配备了 UI 代表时调用。

Swing 提示

把屏幕坐标传送给 `JPopupMenu.setLocation()`

`JPopupMenu` 类提供了两种指定弹出式菜单显示位置的方法：`show (Component c, int x, int`

y) 和 setLocation (int x, int y)。传送给 show 方法的 x, y 坐标指定相对于传送给它的组件 (即弹出式菜单的调用者) 左上角的弹出式菜单的位置。而传送给 setLocation 方法的坐标是屏幕坐标, 即它们是相对于屏幕左上角的坐标。

10.8.7 AWT 兼容

AWT 的弹出式菜单由 java.awt.PopupMenu 类代表, java.awt.PopupMenu 类扩展 java.awt.Menu。另一方面, Swing 的弹出式菜单由 swing.JPopupMenu 类代表, swing.JPopupMenu 类扩展 JComponent。因为 java.awt.Popup 只实现一个自己的方法 show(Component, int, int), 并且 java.awt.Popup 的 show 方法和 JPopupMenu 功能上是相同的, 所以与 AWT 兼容的问题变成了: JPopupMenu 和 java.awt.Menu 是怎样兼容的?

java.awt.Menu 和 JPopupMenu 的 API 在把字符串、菜单项和分隔线添加到菜单中的方式是一样的, 不同之处在于, 对 Swing 弹出式菜单而言, 菜单项添加到 JMenuItem 的一个实例中, 而 java.awt.MenuItem 添加到 AWT 菜单中。

除这两种菜单的显示方式和添加菜单项的方法外, Swing 弹出式菜单与 AWT 对等菜单有明显的差别。首先, AWT 弹出式菜单在显示前, 必须添加到一个组件中, 如下所示:

```
// showing an AWT popup menu

// add popup to this component
add (popup);

//show popup relative to this component
popup.show (this, 10, 10);

//showing a swing popup menu

//show popup relative to this component
popup.show (this, 10, 10);
```

AWT 和 Swing 弹出式菜单另一个明显的差别是可以从 AWT 弹出式菜单中删除菜单项, 但不能从 Swing 弹出式菜单中删除菜单项。另外, AWT 弹出式菜单和 Swing 弹出式菜单都提供对它们所显示的菜单项的访问方法, 然而, 实现这种功能的 API 是不同的。

10.9 JMenuBar

菜单栏是包含一行菜单的容器。Swing 菜单栏可以包含在小应用程序和应用程序中, 而 AWT 菜单栏只能在应用程序中使用。

为了变化一下花样, 图 10-24 所示的菜单栏在应用程序中, 而本章以前所显示的菜单条都在小应用程序中。

例 10-21 列出了图 10-24 所示的应用程序的代码。

例 10-21 一个简单的菜单栏

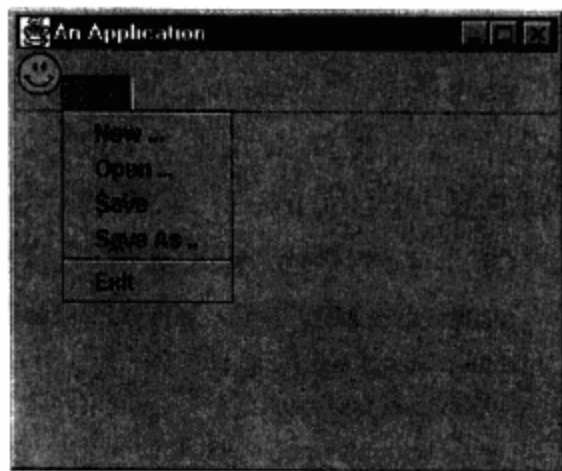


图 10-24 一个应用程序中的菜单栏

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JFrame {
```

```

public Test () {
    JMenuBar mb = new JMenuBar ();
    JMenu fileMenu = new JMenu ("File");
    JMenuItem exitItem = new JMenuItem ("Exit");

    fileMenu.add ("New ...");
    fileMenu.add ("Open ...");
    fileMenu.add ("Save");
    fileMenu.add ("Save As ...");
    fileMenu.addSeparator ();
    fileMenu.add (exitItem);

    mb.add (new JLabel (
        new ImageIcon ("smiley_face_small.gif")));
    mb.add (fileMenu);

    // Either one of the following two lines will
    // attach the menu bar to the application
    //setJMenuBar (mb);
    getRootPane ().setJMenuBar (mb);

    exitItem.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            dispose ();
            System.exit (0);
        }
    });
}

public static void main (String args []) {
    GJApp.launch (new Test (),
        "A Menu Bar", 300, 300, 300, 250);
}
}

```

例 10-21 所列的应用程序除把一个“File”菜单添加到菜单栏中以外，还把一个图标添加到菜单栏中。与所有的 Swing 轻量组件一样，JMenuBar 扩展 JComponent 类，该类又扩展 java.awt.Container。结果，一个菜单栏是一个容器，因此，任何组件都可以添加到 JMenuBar 的实例中。

10.9.1 菜单栏菜单和组件

JMenuBar API 提供了访问包含在菜单栏中的菜单和组件的方法。

JMenuBar 的有些访问方法显示包含在菜单栏中的组件和菜单的信息。

图 10-25 所示的应用程序中的相关代码由与这两个按钮相关联的动作监听器组成。

```

...
menuButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        Component c;
        int cnt = mb.getMenuCount ();
    }
}

```

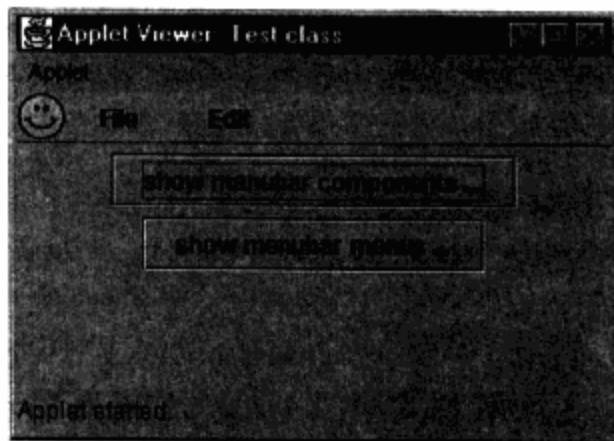


图 10-25 访问菜单栏组件和菜单

```

        for (int i=0; i < cnt; ++i) {
            c = mb.getMenu (i);
            System.out.println (c);
            System.out.println ();
        }
    }
    compButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            Component c;
            int cnt = mb.getComponentCount ();
            for (int i=0; i < cnt; ++i) {
                c = mb.getComponentAtIndex (i);
                System.out.println (c);
                System.out.println ();
            }
        }
    });
}

```

上面所列的第一个动作监听器显示包含在菜单栏中的菜单的信息。这个监听器的 `actionPerformed` 方法调用 `JMenuBar.getMenuCount()` 以计算包含在菜单栏中的菜单数。调用 `JMenuBar.getMenu()` 来获得对每个菜单的引用。

当激活 `menuButton` (菜单按钮) 时, 这个小应用程序显示如下输出:

```

javax.swing.JMenu [, 24, 1, 55x24, alignmentX = null, alignmentY = null, border = javax.swing.plaf.metal.MetalBorders $ MenuItemBorder@13642de, flags = 33, maximumSize = , minimumSize = , preferredSize = , defaultIcon = , disabledIcon = , disabledSelectedIcon = , margin = javax.swing.plaf.Inset $UIResource [top = 2, left = 2, bottom = 2, right = 2], paintBorder = true, paintFocus = false, pressedIcon = , rolloverEnabled = false, rolloverIcon = , rolloverSelectedIcon = , selectedIcon = , text = File]

javax.swing.JMenu [, 79, 1, 57x24, alignmentX = null, alignmentY = null, border = javax.swing.plaf.metal.MetalBorders $ MenuItemBorder@13642de, flags = 33, maximumSize = , minimumSize = , preferredSize = , defaultIcon = , disabledIcon = , disabledSelectedIcon = , margin = javax.swing.plaf.Inset $UIResource [top = 2, left = 2, bottom = 2, right = 2], paintBorder = true, paintFocus = false, pressedIcon = , rolloverEnabled = false, rolloverIcon = , rolloverSelectedIcon = , selectedIcon = , text = Edit]

```

通过调用 `java.awt.Container` 中类的 `getComponentCount()` 和 `JMenuBar.getComponentAtIndex()`, 上面所列的第二个动作监听器显示包含在菜单栏中的组件的有关信息。

当激活 `compButton` (组件按钮) 时, 这个小应用程序显示下面的输出:

```

javax.swing.JLabel [, 0, 1, 24x24, alignmentX = 0.0, alignmentY = null, border = , flags = 0, maximumSize = , minimumSize = , preferredSize = , defaultIcon = javax.swing.ImageIcon@9bc242de, disabledIcon = horizontalAlignment = CENTER, horizontalTextPosition = RIGHT, iconTextGap = 4, labelFor = , text = , verticalAlignment = CENTER, verticalTextPosition = CENTER]

javax.swing.JMenu [, 24, 1, 55x24, alignmentX = null, alignmentY = null, border = javax.swing.plaf.metal.MetalBorders $ MenuItemBorder@13642de, flags = 33, maximumSize = , minimumSize = , preferredSize = , defaultIcon = , disabledIcon = , disabledSelectedIcon = , margin = javax.swing.plaf.Inset

```



```
sUIResource [ top = 2, left = 2, bottom = 2, right = 2], paintBorder = true, paintFocus = false, pressedIcon = , rolloverEnabled = false, rolloverIcon = , rolloverSelectedIcon = , selectedIcon = , text = File]

javax.swing.JMenu [ , 79, 1, 57x24, alignmentX = null, alignmentY = null, border = javax.swing.plaf.metal.MetalBorders $ MenuItemBorder@13642de, flags = 33, maximumSize = , minimumSize = , preferredSize = , defaultIcon = , disabledIcon = , disabledSelectedIcon = , margin = javax.swing.plaf.Inset
sUIResource [ top = 2, left = 2, bottom = 2, right = 2], paintBorder = true, paintFocus = false, pressedIcon = , rolloverEnabled = false, rolloverIcon = , rolloverSelectedIcon = , selectedIcon = , text = Edit]
```

例 10-22 列出了图 10-25 所示的小应用程序的代码。

例 10-22 菜单栏中的菜单和组件

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();

        final JMenuBar mb = new JMenuBar ();
        JMenu fileMenu = new JMenu ("File");
        JMenu editMenu = new JMenu ("Edit");
        JMenuItem exitItem = new JMenuItem ("Exit");
        JButton compButton = new JButton (
            "show menubar components ...");
        JButton menuButton = new JButton (
            "show menubar menus ...");

        fileMenu.add ("New ...");
        fileMenu.add ("Open ...");
        fileMenu.add ("Save");
        fileMenu.add ("Save As ...");
        fileMenu.addSeparator ();
        fileMenu.add (exitItem);

        editMenu.add ("Undo");
        editMenu.addSeparator ();
        editMenu.add ("Cut");
        editMenu.add ("Copy");
        editMenu.add ("Paste");

        mb.setMargin (new Insets (30, 20, 10, 5));
        mb.add (new JLabel (new ImageIcon ("smiley.gif")));
        mb.add (fileMenu);
        mb.add (editMenu);

        setJMenuBar (mb);
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (compButton);
        contentPane.add (menuButton);

        exitItem.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                System.exit (0);
            }
        });
    }
}
```




图 10-26 JMenuBar 类图

菜单之间的边距。JMenuBar 还维护对一个 SingleSelectionModel 的一个 private 引用，SingleSelectionModel 用于跟踪当前选取的菜单。

10.9.2 JMenuBar 属性

表 10-7 列出了由 JMenuBar 的实例所维护的属性。

表 10-7 JMenuBar 属性

属性名	数据类型	访问 ^②	类型 ^③	缺省 ^④
borderPainted	boolean	SG	S	true
componentAtIndex	Component	G	I	—
componentIndex	int	G	S	—
helpMenu ^①	JMenu	SG	S	—
margin	int	SG	S	L&F
menuCount	int	G	S	—
selectionModel	SingleSelectionModel	SG	S	DefaultSingleSelectionModel

① 在 Swing 1.1 中，没有实现帮助菜单
② C = 可在创建时设置/G = 获取方法/S = 设置方法
③ B = 关联的/bool = 布尔 /C = 受约束的/I = 索引的/S = 简单的/RO = 只读
④ L&F = 与界面样式有关

borderPainted——确定是否绘制菜单栏的边框；缺省时，绘制边框。borderPainted 属性可能被有些界面样式所忽略。

componentAtIndex——一个计算属性，它返回给定索引的一个组件。添加到菜单栏的第一个组件的索引为 0，添加到菜单栏中的最后一个组件的索引值是 `getComponentCount()` -1。

我们知道，任何类型的组件都可以被添加到一个菜单栏中，因为 Swing 轻量组件都是 AWT 容器。因此，不能假定从 `getComponentAtIndex()` 返回的组件是一个菜单。

componentIndex——返回包含在一个菜单栏中的给定组件的索引值。添加到菜单栏的第一个组件的索引为 0，添加到菜单栏中的最后一个组件的索引值是 `getComponentCount()` -1。

helpMenu——一个帮助菜单，在菜单栏中它的代表是与界面样式有关。在 Swing 1.1 FCS 中，没有实现帮助菜单，换句话说，在 Swing 1.1 FCS 下设置 help Menu 属性是没有效果的。

menuCount——`menuCount` 属性不像期望的那样代表菜单栏中的菜单数。这个属性代表菜单栏中所包含的组件数。

selectionModel——缺省情况下，菜单栏有一个 `DefaultSingleSelectionModel` 属性，该属性维护菜单栏中当前选取的菜单元素。

10.9.3 JMenuBar 事件

除在关联属性修改时激发属性变化事件外，`JMenuBar` 没有自己需要激发的事件。

10.9.4 JMenuBar 类总结

类总结 10-7 列出了 `JMenuBar` 的 `public` 和 `protected` 的变量和方法。

类总结 10-7 JMenuBar

扩展：`JComponent`

实现：`MenuElement`、`java.accessibility.Accessible`

1. 构造方法

`public JMenuBar()`

上面所列的无参数构造方法是唯一构造一个 `JMenuBar` 实例的方法。构造 `JMenuBar` 实例后，可以通过下面所列的方法可以把菜单和组件添加到 Swing 菜单栏中。

2. 方法

(1) 菜单

`public JMenu add(JMenu)`

`public JMenu getMenu(int index)`

`public int getMenuCount()`

`public JMenu getHelpMenu()`

`public void setHelpMenu(JMenu)`

用 `JMenuBar.add` 方法可以把菜单添加到一个菜单栏中，然而，`JMenuBar` 类没有提供从一个菜单栏中删除菜单的相应方法。要从 `JMenuBar` 删除一个菜单，则要使用 `remove(int index)` 方法，这个方法是 `JMenuBar` 从 `java.awt.Container` 继承来的。

在 Swing 1.1 FCS 中，`getMenuCount` 方法返回菜单栏中包含的组件数（不是菜单数）。`getMenu` 方法以一个组件索引为参数，例如，如果一个菜单栏包含一个组件和两个菜单（按这个顺序添加到菜单栏中），调用 `getMenu(1)` 将获得对第一个菜单的一个引用。

在 Swing 1.1 FCS 中，设置帮助菜单没有效果，而且上面所列的最后两个方法将弹出一个错误信息，指示这些方法还没有实现。

(2) 组件

```

public Component getComponentAtIndex (int index)
public int getComponentIndex (Component)
public void setSelected (Component)
public boolean isSelected ()

```

可以用上面所列的方法来确定菜单栏中所包含的组件和它们的索引。给定一个有效索引值，`getComponentAtIndex` 方法返回对一个组件的引用。如果该索引是无效的，则返回 `null` 引用。`getComponentIndex` 返回给定组件的一个索引，如果组件没有包含在菜单栏中，则返回 -1。

`setSelected` 方法使用菜单栏的选取模型来选取指定的组件。

(3) 边距/选取模型/边框绘制/管理焦点

```

public Insets getMargin ()
public void setMargin (Insets)

public SingleSelectionModel getSelectionModel ()
public void setSelectionModel (SingleSelectionModel)

public boolean isBorderPainted ()
public void setBorderPainted (boolean)
protected void paintBorder (Graphics)

public boolean isManagingFocus ()

```

菜单栏边距定义为菜单栏的组件和菜单栏的边框之间的空间。有些界面样式（包括 Metal 界面样式）忽略菜单栏的边距。

为菜单栏的选取模型提供了访问方法，这个模型是 `SingleSelectionModel` 的一个实例。可以指定 `borderPainted` 属性，然而，与 `margin` 属性一样，某些界面样式可能忽略 `borderPainted` 属性。

(4) MenuElement 接口

```

public abstract Component getComponent ()
public abstract MenuElement [] getSubElements ()

public abstract void menuSelectionChanged (boolean)
public abstract void processKeyEvent (KeyEvent, MenuElement [], MenuSelectionManager)
public abstract void processMouseEvent (MouseEvent, MenuElement [], MenuSelectionManager)

```

上面所列的这些方法在 `MenuElement` 接口中定义。有关 `MenuElement` 接口的详细内容，请参见 10.7 节“MenuElement”。

`getComponent` 方法返回对菜单栏本身的一个引用。`getSubElements()` 方法返回菜单栏中所包含的菜单元素的一个数组。注意，从 `getSubElements()` 返回的 `MenuElement` 数组可以为 `null`，因为 `getSubElements()` 使用 `JMenuBar.getMenu()`，它返回菜单栏组件（不是菜单）的 `null` 引用。

(5) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public void updateUI ()
public MenuBarUI getUI ()
public void setUI (MenuBarUI)

```

上面的方法可以在大多数 `JComponent` 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。`updateUI` 方法在组件配备了 UI 代表时调用。

10.9.5 AWT 兼容

表 10-8 列出了由 `java.awt.MenuBar` 实现的 `public` 方法和由 `JMenuBar` 实现的对应方法。

表 10-8 java.awt.MenuBar 方法和 JMenuBar 的对应方法^①

java.awt.MenuBar 方法	JMenuBar 的对应方法
add (Menu)	add (JMenu)
delecteShortcut (MenuShortcut)	——
getHelpMenu ()	getHelpMenu ()
getMenu (int)	getMenu (int)
getMenuCount ()	getMenuCount ()
getShortcutMenuItem (MenuShortcut)	——
remove (int)	remove (int)
remove (MenuComponent)	——
setHelpMenu (Menu)	setHelpMenu (JMenu)
shortcuts ()	——

① 对具有不同原型的方法用黑体字显示。

通过 AWT 的 MenuBar 类可以把 AWT 菜单的捷径键分配给菜单项。把捷径键分配给 Swing 菜单项不是通过 JMenuBar 类来处理的，相反，捷径键必须以快捷键的形式直接分配给菜单项。

10.10 JToolBar

当今的用户界面没有不使用工具条的。在 Swing 中，工具条最终会成为 JDK 的一部分。

Swing 工具条由 JToolBar 类来表示，该类本质上是一个具有水平或垂直方向的容器。Swing 工具条可以浮动，可以把它们拖到一个容器的北边、南边、东边或西边，或拖到一个单独的窗口中。除通过调用 JToolBar add (Action) 和 addSeparator 方法把动作和分隔线添加到工具条中外，还可以把任何类型的组件添加到工具条中。与菜单栏不同，在 Swing 顶层容器（如 JApplet 和 JRootPane）中没有添加工具条的特定方法。除应该用 BorderLayout 的一个实例把工具条添加到一个容器中作为容器中的北边、南边或西边的组件外，把工具条添加到容器中的方式与添加任何 Swing 组件的方式相同。

图 10-27 所示的小应用程序把五个按钮和两个组合框添加到一个工具条中。这个小应用程序接着把这个工具条添加到这个小应用程序的内容窗格中作为北边组件。

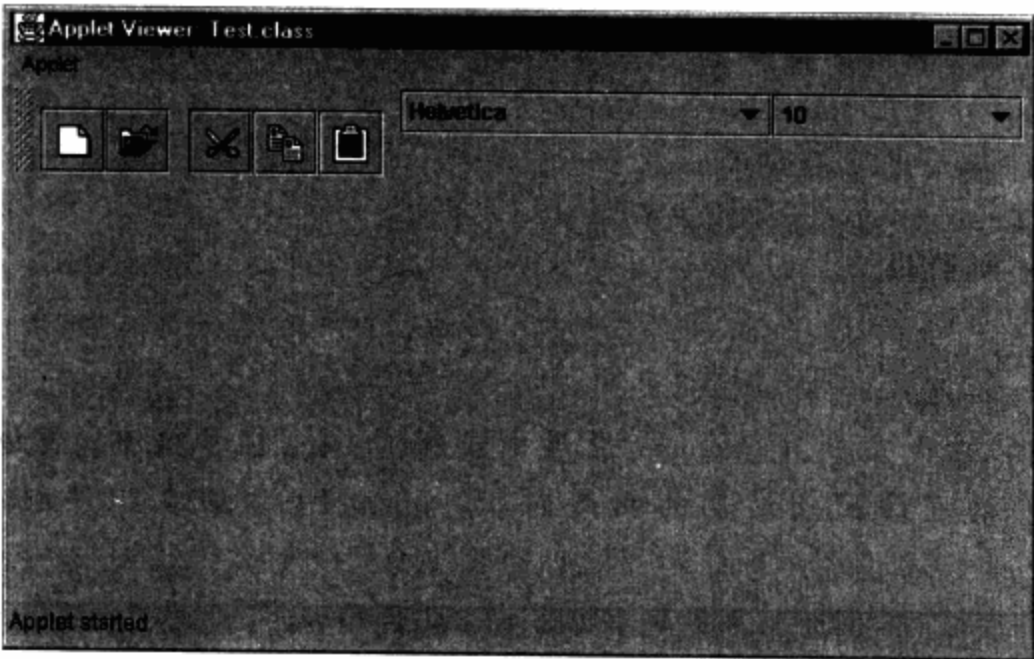


图 10-27 把组件添加到一个工具条中

图 10-27 所示的小应用程序的源代码很简单，其完整的代码列在例 10-23 中。

例 10-23 把组件添加到一个菜单栏中

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JToolBar tb = new JToolBar ();
        JComboBox fontCombo = new JComboBox (),
            fontSizeCombo = new JComboBox ();

        JButton newButton = new JButton (new ImageIcon ("new.gif")),
            openButton = new JButton (new ImageIcon ("open.gif")),
            cutButton = new JButton (new ImageIcon ("cut.gif")),
            copyButton = new JButton (new ImageIcon ("copy.gif")),
            pasteButton = new JButton (new ImageIcon ("paste.gif"));

        fontCombo.addItem ("Helvetica");
        fontCombo.addItem ("Palatino");
        fontCombo.addItem ("Courier");
        fontCombo.addItem ("Times");
        fontCombo.addItem ("Times-Roman");

        fontSizeCombo.addItem ("10");
        fontSizeCombo.addItem ("12");
        fontSizeCombo.addItem ("14");
        fontSizeCombo.addItem ("16");
        fontSizeCombo.addItem ("18");

        tb.add (newButton);
        tb.add (openButton);

        tb.addSeparator ();

        tb.add (cutButton);
        tb.add (copyButton);
        tb.add (pasteButton);

        tb.addSeparator ();

        tb.add (fontCombo);
        tb.add (fontSizeCombo);

        contentPane.setLayout (new BorderLayout ());
        contentPane.add (tb, BorderLayout.NORTH);
    }
}
```

注意 这个小应用程序内容窗格的布局管理器设置为 BorderLayout 的一个实例，因为，缺省情况下，小应用程序的内容窗格的布局管理器是 FlowLayout 的一个实例。

还要注意的，图 10-27 所示的小应用程序中的按钮和组合框布局不美观。按钮和组合框的垂直中心没有在一条线上，而且组合框比需要的大。导致这种现象的原因有两个：

首先，JToolBar 的实例使用一个 BoxLayout 实例来布局它们所包含的组件。由于对组合框的

最大尺寸没有限制，所以组合框被拉伸以填充工具条中剩余的可使用空间。

其次，BoxLayout 根据组件的 X 和 Y 排列来排列组件（有关 BoxLayout 的详细内容，请参见 6.5.1 节“BoxLayout 类”）。当水平地布局组件时，BoxLayout 根据组件的 Y 排列来排列组件的垂直中心。缺省情况下，组合框的 Y 排列是 0.5，而按钮的 Y 排列是 0.0。

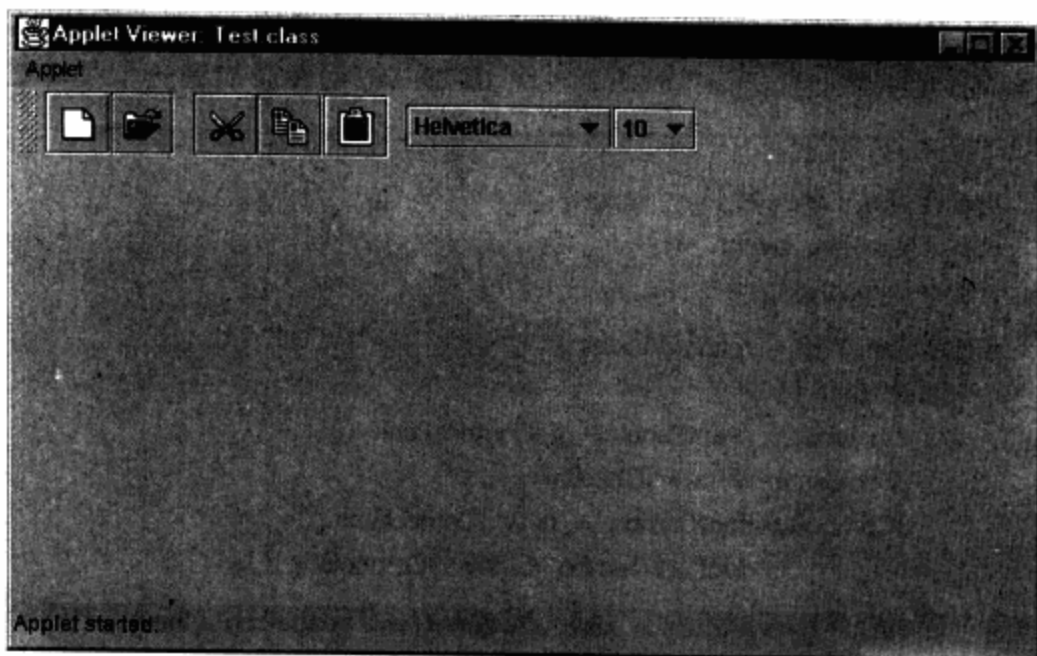


图 10-28 把组件添加到一个工具条中（方法 2）

图 10-28 示出了图 10-27 所示的小应用程序的另一个版本，在这里，把工具条中的组件排成了一行，而且组合框的大小也比较合适。

例 10-24 列出了图 10-28 所示的小应用程序的代码。

例 10-24 把组件添加到一个工具条中（方法 2）

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JToolBar tb = new JToolBar ();
        JComboBox fontCombo = new JComboBox (),
            fontSizeCombo = new JComboBox ();

        JButton newButton = new JButton (new ImageIcon ("new.gif")),
            openButton = new JButton (new ImageIcon ("open.gif")),
            cutButton = new JButton (new ImageIcon ("cut.gif")),
            copyButton = new JButton (new ImageIcon ("copy.gif")),
            pasteButton = new JButton (new ImageIcon ("paste.gif"));

        fontCombo.addItem ("Helvetica");
        fontCombo.addItem ("Palatino");
        fontCombo.addItem ("Courier");
        fontCombo.addItem ("Times");
        fontCombo.addItem ("Times-Roman");

        fontSizeCombo.addItem ("10");
        fontSizeCombo.addItem ("12");
        fontSizeCombo.addItem ("14");
        fontSizeCombo.addItem ("16");
        fontSizeCombo.addItem ("18");

        tb.add (newButton);
        tb.add (openButton);
        tb.addSeparator ();
        tb.add (cutButton);
```

```

tb.add (copyButton);
tb.add (pasteButton);

tb.addSeparator ();

tb.add (fontCombo);
tb.add (fontSizeCombo);

newButton.setAlignmentY (0.5f);
openButton.setAlignmentY (0.5f);
cutButton.setAlignmentY (0.5f);
copyButton.setAlignmentY (0.5f);
pasteButton.setAlignmentY (0.5f);

newButton.setAlignmentX (0.5f);
openButton.setAlignmentX (0.5f);
cutButton.setAlignmentX (0.5f);
copyButton.setAlignmentX (0.5f);
pasteButton.setAlignmentX (0.5f);

fontCombo.setMaximumSize (fontCombo.getPreferredSize ());
fontSizeCombo. setMaximumSize (
    fontSizeCombo.getPreferredSize ());

contentPane.setLayout (new BorderLayout ());
contentPane.add (tb, BorderLayout.NORTH);

```

把按钮的排列设置为与组合框的排列相匹配，而且把组合框的最大尺寸设置为组合框的首选大小。

结果，根据按钮和组合框的垂直中心来排列这些按钮和组合框，而且组合框的大小不比它们的首选大小宽。注意，按钮的 X 和 Y 排列都设置为与组合框的 X 和 Y 排列相匹配，以便当把工具条拖动为垂直方向排列时，按钮和组合框将根据它们的水平中心来排列。

10.10.1 滚过式工具条

有些界面样式对 `isRollover` 工具条客户属性是敏感的。把工具条的这个属性设置为 `true` 可能导致只有当光标进入按钮上时才显示工具条按钮的边框。图 10-29 显示了滚过式的效果。

例 10-25 列出了图 10-29 所示的小应用程序的代码。

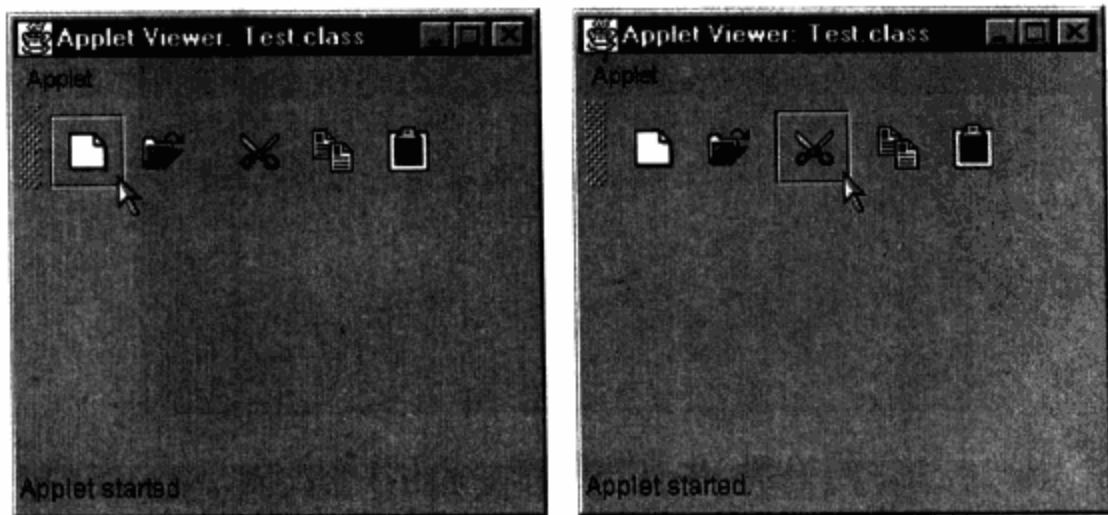


图 10-29 指定 `JToolBar.isRollover` 客户属性

例 10-25 JToolBar.isRollover 属性

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane () ;
        JToolBar toolbar = new JToolBar () ;

        toolbar.add (new NewAction ()) ;
        toolbar.add (new OpenAction ()) ;
        toolbar.addSeparator () ;
        toolbar.add (new CutAction ()) ;
        toolbar.add (new CopyAction ()) ;
        toolbar.add (new PasteAction ()) ;

        toolbar.putClientProperty ("JToolBar.isRollover",
                                   Boolean.TRUE) ;

        contentPane.add (toolbar, BorderLayout.NORTH) ;
    }
    // Action class listings omitted:
    // see "A Menu Bar and a Toolbar in a JRootPane" on Page 455
}
```

在 Swing 1.1 FCS 中, isRollover 客户属性只有在 Java 界面样式中才会有效果。

10.10.2 在工具条中使用动作

与菜单一样, 可以把动作添加到工具条中。JToolBar.add (Action) 方法提取与动作相关联的图标和文本, 并创建一个适当的按钮。

图 10-30 展示了另一个不美观的工具条, 它把只有图像的按钮和包含图像和文本的按钮混合起来。

例 10-26 列出了图 10-30 所示的小应用程序的代码。

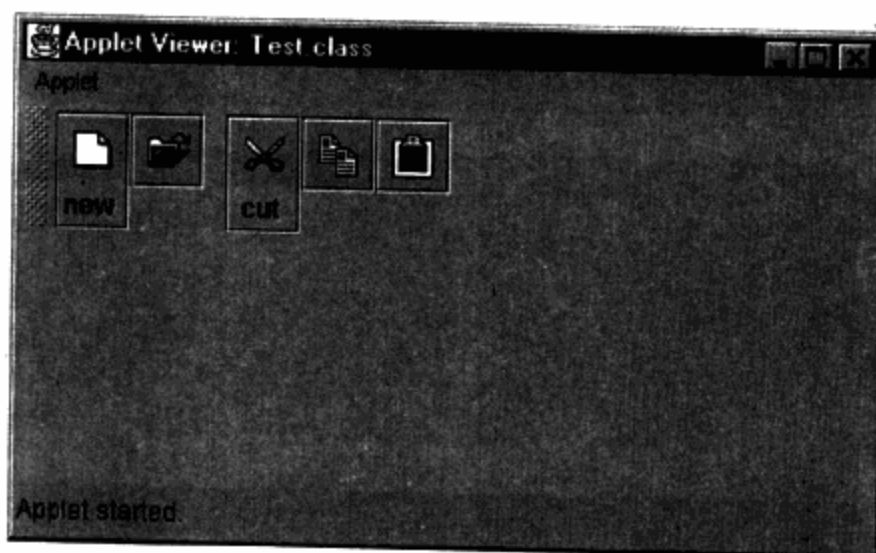


图 10-30 把动作添加到一个工具条中

例 10-26 把动作添加到一个工具条中

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane () ;
        JToolBar toolbar = new JToolBar () ;

        toolbar.add (new NewAction ()) ;
        toolbar.add (new OpenAction ()) ;
        toolbar.addSeparator () ;
        toolbar.add (new CutAction ()) ;
        toolbar.add (new CopyAction ()) ;
        toolbar.add (new PasteAction ()) ;

        contentPane.add (toolbar, BorderLayout.NORTH) ;
    }

    class NewAction extends AbstractAction {
        public NewAction () {
            super ("new", new ImageIcon ("new.gif")) ;
        }

        public void actionPerformed (ActionEvent event) {
            showStatus ("new") ;
        }
    }

    class OpenAction extends AbstractAction {
        public OpenAction () {
            putValue (Action.SMALL_1_ICON,
                new ImageIcon ("open.gif")) ;
        }

        public void actionPerformed (ActionEvent event) {
            showStatus ("open") ;
        }
    }

    class CutAction extends AbstractAction {
        public CutAction () {
            super ("cut", new ImageIcon ("cut.gif")) ;
            putValue (Action.SMALL_1_ICON, new ImageIcon ("cut.gif")) ;
        }

        public void actionPerformed (ActionEvent event) {
            showStatus ("cut") ;
        }
    }

    class CopyAction extends AbstractAction {
        public CopyAction () {
            putValue (Action.SMALL_1_ICON,
                new ImageIcon ("copy.gif")) ;
        }

        public void actionPerformed (ActionEvent event) {
            showStatus ("copy") ;
        }
    }
}
```

```

class PasteAction extends AbstractAction {
    public PasteAction () {
        putValue (Action.SMALL_ICON,
            new ImageIcon ("paste.gif"));
    }
    public void actionPerformed (ActionEvent event) {
        showStatus ("paste");
    }
}

```

这个小应用程序相当长，但是其中的大部分程序是与实现工具条按钮的动作有关的。在 Swing 1.1 FCS 中，没有办法压制与工具条中的一个动作相关联的文本的显示^①。这是很遗憾的，因为工具条按钮通常不显示文本，使用动作的益处是多个组件可共享它们。因为菜单通常显示文本，所以在菜单和工具条菜单之间共享动作意味着必须把与这个动作相关联的文本显示在工具条按钮中。

10.10.3 浮动工具条

如前所述，Swing 工具条可以浮动，如图 10-31 所示。图 10-31 所示的小应用程序与图 10-26 所示的小应用程序相同，所以在此就不列出来了。

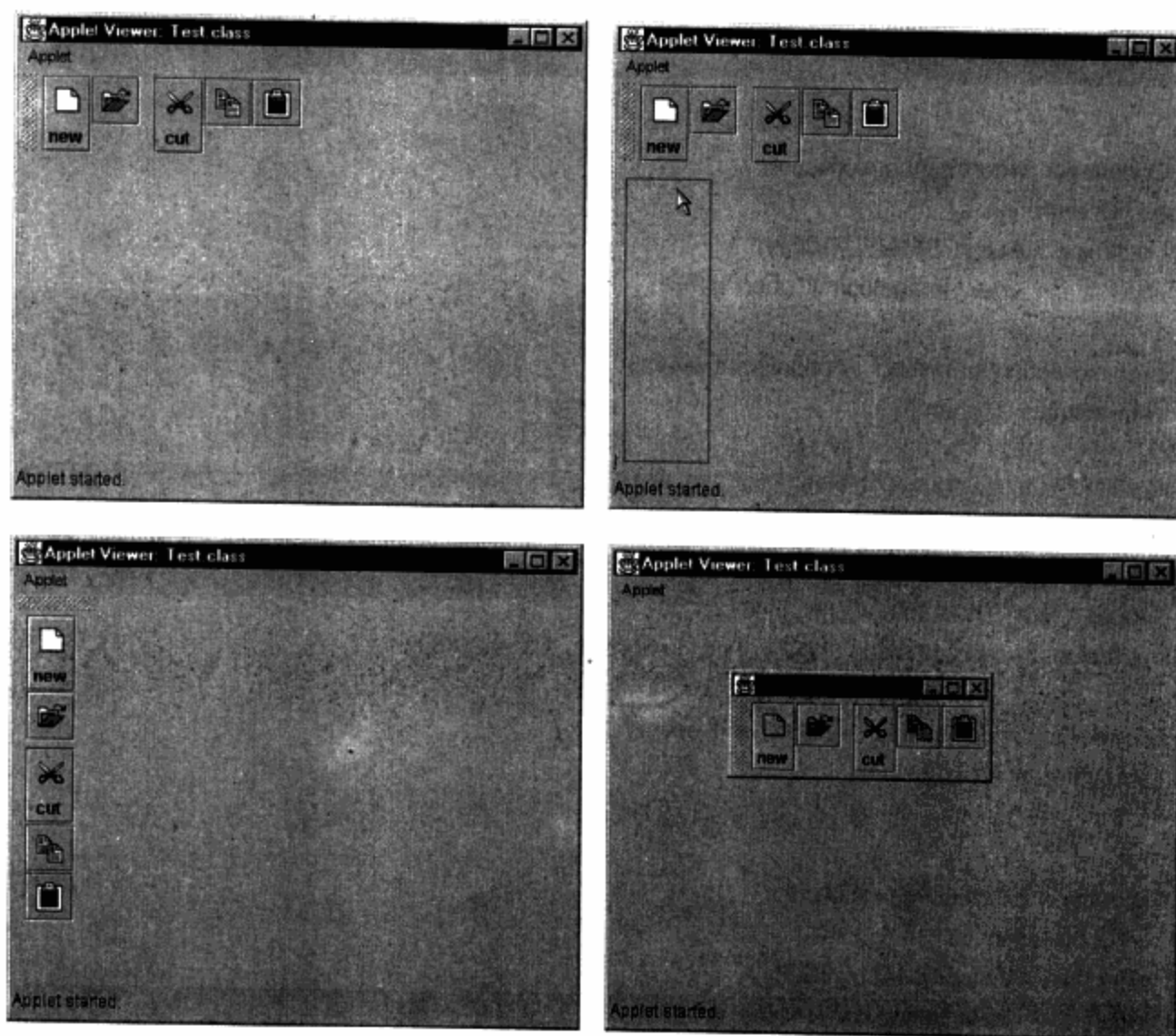


图 10-31 一个浮动的工具条

① 如果工具条是手工创建的，则可以压制文本的显示——参见例 22-11。

图 10-31 左上图显示这个小应用程序的初始样子。右上图显示拖动工具条时的样子。左下图显示把工具条放在这个小应用程序的内容窗格的西边后的样子。右下图显示在工具条完全拖出窗口后的样子。当工具条拖出它原来被包含的窗口时，它放在单独的窗口中。

缺省情况下，工具条是浮动的（参见表 10-9）。可以以 `false` 为参数来调用 `JToolBar.setFloatable` 方法，以便使工具条不浮动。

10.10.4 位置固定的工具提示

通常把与工具条按钮相关联的工具提示显示在相对于这个按钮的固定位置上。通常，把工具提示直接显示在这个按钮下面。图 10-32 所示的小应用程序包含以这种方式显示工具提示的按钮。

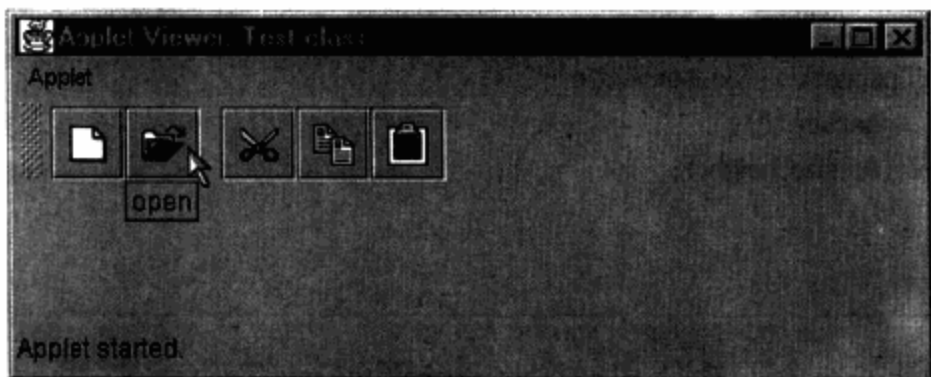


图 10-32 在相对于工具条按钮的固定位置上显示工具提示

例 10-27 列出了图 10-32 所示的小应用程序的代码。

例 10-27 固定位置的工具提示

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JToolBar toolbar = new JToolBar ();

        String [] tooltipTexts = {"new", "open", "cut", "copy",
                                   "paste"};

        ImageIcon [] icons = {
            new ImageIcon ("new.gif"),
            new ImageIcon ("open.gif"),
            new ImageIcon ("cut.gif"),
            new ImageIcon ("copy.gif"),
            new ImageIcon ("paste.gif")
        };

        JButton [] buttons = {
            new ButtonWithFixedTooltip (icons [0], tooltipTexts [0]),
            new ButtonWithFixedTooltip (icons [1], tooltipTexts [1]),
            new ButtonWithFixedTooltip (icons [2], tooltipTexts [2]),
            new ButtonWithFixedTooltip (icons [3], tooltipTexts [3]),
            new ButtonWithFixedTooltip (icons [4], tooltipTexts [4])
        };
    }
}
```

```
for (int i=0; i < buttons.length; ++i) {
    toolbar.add (buttons [i]);

    if (tooltipTexts [i] .equals ("open"))
        toolbar.addSeparator ();
}

contentPane.add (toolbar, BorderLayout.NORTH);
}

class ButtonWithFixedTooltip extends JButton {
public ButtonWithFixedTooltip (Icon icon,
                               String tooltipText) {
    super (icon);
    setToolTipText (tooltipText);
}

public Point getToolTipLocation (MouseEvent e) {
    Dimension size = getSize ();
    return new Point (0, size.height);
}
}
```

这个小应用程序创建一个工具条和一个 ButtonWithFixedTooltips 数组，然后把按钮添加到工具条中。ButtonWithFixedTooltips 类是 JButton 的一个简单扩展，它重载 getToolTipLocation 方法以返回直接在按钮下面的位置。在 4.7.1 节“基于鼠标位置的工具体提示”中对指定工具体提示的位置有详细的介绍。组件总结 10-7 总结了 JToolBar 类。

组件总结 10-7 JToolBar

模型： ——
UI 代表： javax.swing.plaf.basic.BasicToolBarUI

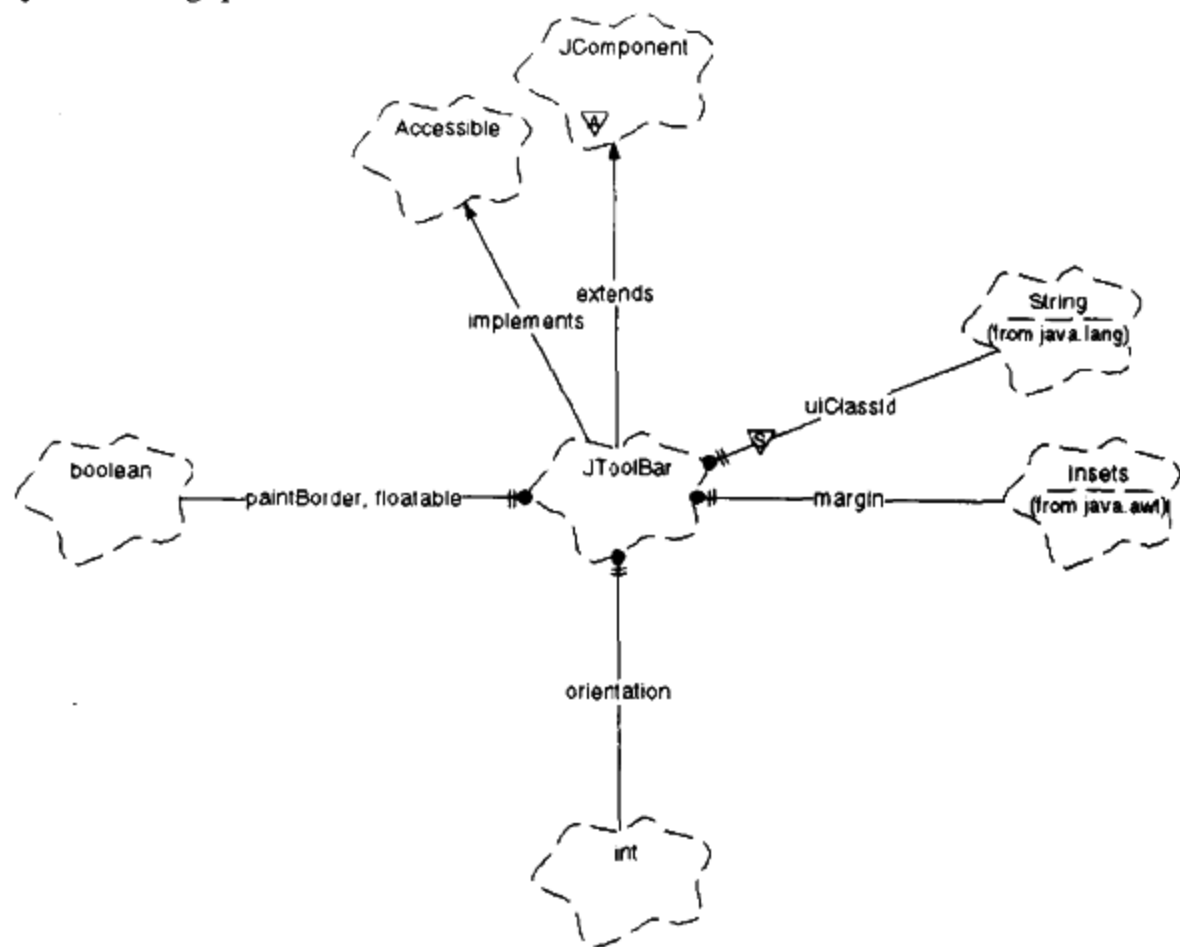


图 10-33 JToolBar 类图

绘制器： ——
编辑器： ——
激发的事件： PropertyChangeEvent
替换： ——
类图： 见图 10-33

JToolBar 是 JComponent 的一个简单扩展，它本质上是显示一行按钮或一系列按钮的容器。

JToolBar 可设置对一些 boolean 值的引用，这些值帮助 JToolBar 跟踪是否绘制边框和工具条是否是可浮动。工具条可以浮动指可以把工具条拖动到容器中的指定位置或把工具条拖动到一个单独的窗口中。工具条还可以确定工具条边框与它的按钮之间的边距。

10.10.5 JToolBar 属性

表 10-9 列出了由 JToolBar 类维护的属性。

表 10-9 JToolBar 属性

属性名	数据类型	访问 ^①	类型 ^②	缺省值 ^③
borderPainted	boolean	SG	B	true
componentIndex	int	G	S	——
componentAtIndex	Component	G	I	——
floatable	boolean	SG	B	true
orientation	int	SG	B	HORIZONTAL
margin	int	SG	B	(0, 0, 0, 0)

① C = 可在创建时设置/G = 获取方法/S = 设置方法
② B = 关联的（激发 PropertyChangeEvent）/C = 受约束的/I = 索引的/
S = 简单的/Ch = 激发 ChangeEvent
③ L&F = 与界面样式有关

borderPainted——确定是否要绘制工具条的边框。

componentIndex——代表工具条中指定组件的索引。如果该组件没有包含在工具条中，则从 getComponentIndex 方法返回-1。

componentAtIndex——代表工具条中的特定位置上的一个组件。如果传送给 getComponentAtIndex 方法的索引是无效的，则该方法返回一个 null 索引。

当确定传送给 getComponentAtIndex 方法的索引时，必须把分隔线考虑进去。例如，假如一个工具条包含一个组件后跟一个分隔线和另一个组件，如果要获得对工具条第二个组件的一个引用，则索引值是 2。

floatable——代表是否可以把一个工具条拖动到一个窗口的新的位置上或把它拖动到它自己的一个窗口中。

orientation——可以把工具条水平放置或垂直放置，Orientation 属性是 JToolBar.HORIZONTAL 或 JToolBar.VERTICAL。如果传送这两个常量以外的数值，则 JToolBar.setOrientation 方法将弹出一个异常信息指出参数是非法参数。

margin——代表工具条边框的内边缘与工具条组件的外边缘之间的边距。如果该边距设置为 null（这是缺省值），则工具条使用缺省边距 (0, 0, 0, 0)。

如果边距设置为非 null 的边衬，则工具条的缺省边框使用这个边距值来产生工具条边框和它的组件之间的间隙。如果为工具条显式地设置了边框，则该边框负责把工具条的边距考虑进

去。

10.10.6 JToolBar 事件

除在 JToolBar 的关联属性修改时激发属性变化事件外, JToolBar 类不激发它自己的事件。

10.10.7 JToolBar 类总结

类总结 10-8 列出了 JToolBar 的 public 和 protected 变量和方法。

类总结 10-8 JToolBar

扩展: JComponent

实现: SwingConstants、javax.accessibility.Accessible

1. 构造方法

public JToolBar ()

JToolBar 只提供一个无参数的构造方法。在构造后, 通过调用 add 或 add (Action) 方法, 可以把按钮添加到工具条中。

2. 方法

(1) 方向

public int getOrientation ()

public void setOrientation (int orientation)

上面所列的这两个方法是工具条方向属性的访问方法。setOrientation 方法的有效值是 JToolBar.HORIZONTAL 和 JToolBar.VERTICAL。如果 setOrientation 以一个无效的 integer 值为参数, 则将弹出一个异常信息。

动作变化监听器

protected PropertyChangeListener createActionChangeListener (JButton)

createActionChangeListener 方法创建与一个按钮相关联的动作监听器, 这个按钮是由一个动作创建的。缺省时, 这个监听器是 ActionListener 的一个实例, ActionListener 是 JToolBar 类的一个私有类。这个监听器响应创建这个按钮的动作。如果与这个动作相关联的名字或小图标修改了, 则更新这个按钮。

如果由 ActionListener 类提供的缺省行为不能满足需要, 则 JToolBar 的扩展可以重载 createActionChangeListener 方法, 但是实际上, 很少重载这个方法。

(2) 添加/删除

public JButton add (Action)

public void addSeparator ()

public void addSeparator (Dimension size)

public void remove (Component)

上面所列的这些方法分别把按钮和分隔线添加到一个工具条中。注意, 这里的分隔线与添加到菜单中的分隔线不同。添加到菜单中的分隔线绘制一根蚀刻线, 而添加到工具条中的分隔线只提供工具条组件之间的空白空间。

(3) 属性访问方法

public Component getComponentAtIndex (int)

public int getComponentIndex (Component)

public Insets getMargin ()

public void setBorderPainted (boolean)

```

public void setFloatable (boolean)
public void setMargin (Insets)

public boolean isBorderPainted ()
public boolean isFloatable ()
protected void paintBorder (Graphics)

```

上面所列的这些方法是表 10-9 所列的属性的访问方法。

(4) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public ToolBarUI getUI ()
public String getUIClassID ()
public void setUI (ToolBarUI)
public void updateUI ()

```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

10.10.8 AWT 兼容

AWT 没有提供工具条组件。

10.11 本章回顾

Swing 的菜单系统在 AWT 菜单系统基础上做了很大的改进。Swing 菜单和菜单项都是按钮和容器，由于 AbstractButton 和 java.awt.Container 分别是它们的超类。菜单和菜单项可以配备文本和（或）一个图标，也可以包含任意的组件。有关 Swing 组件包含组件的能力的详细内容，请参见第 4 章“JComponent 类”。

Swing 弹出式菜单是独特的，因为它们可以在轻量、中量和重量三种不同类型的容器中显示。这三种弹出式菜单的类型术语可能有些混乱，因为中量弹出式菜单实际上意味着弹出式菜单在一个重量 AWT 面板中显示；而重量弹出式菜单则在一个本地窗口中显示。但是，控制显示弹出式菜单的容器类型是非常重要的。

与弹出式菜单一样，工具条是基本的用户界面组件，这是 AWT 所缺乏的。与菜单和菜单项一样，Swing 工具条最终是 java.awt.Container 的一个扩展，因此，可以把任何组件添加到一个工具条中。

第 11 章 进度条、滑杆和分隔条

本章介绍三个 Swing 组件：JProgressbar、JSlider 和 JSeparator。

人们经常用进度条反映耗时任务已完成的百分比，本章将提供这样一个例子。但是，用 ProgressMonitor 和 ProgressMonitorInputStream 实用工具来反映一个任务的进度信息通常会更容易。“进度监控”小节中讨论了 ProgressMonitor 和 ProgressMonitorInputStream 实用工具。

JProgressbar 和 JSlider 与 JSeparator 一样，用来描述一个介于最小值和最大值之间的值。因此，与 JProgressbar 和 JSlider 相关的模型是 BoundedRangeModel 接口的一个实现。

JSeparator 是绘制一条蚀刻线的组件，它把逻辑上相关的组件分隔在一组，JSeparator 组件可水平地或垂直地放置。

11.1 JProgressbar

JProgressbar 是一个简单的组件，它一般是一种颜色部分或完全填充的矩形。缺省情况下，进度条配备了一个凹陷的边框，并水平放置。

进度条还可选择显示一个字符串，这个字符串在进度条矩形的中央位置上显示。这个字符串缺省时为耗时任务已完成的百分比。这个字符串可用 JProgressbar.setString 方法定制。

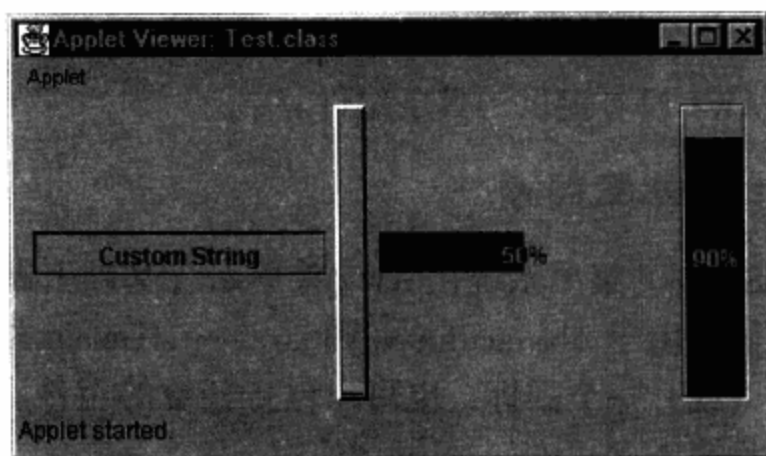


图 11-1 各种配置的进度条

图 11-1 中示出的小应用程序创建了不同配置的进度条。

例 11-1 列出了图 11-1 中示出的小应用程序的代码。

例 11-1 各种配置的进度条

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private JProgressbar [] progressBars = {
        new JProgressbar (),
        new JProgressbar (),
        new JProgressbar (),
        new JProgressbar ()
    };

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        for (int i=0; i < progressBars.length; ++i) {
```

```

JProgressBar pb = progressBars [i];
if (i == 0) {
    pb.setStringPainted (true);
    pb.setString (" Custom String");
}
if (i == 1) {
    pb.setOrientation (JProgressBar.VERTICAL);
    pb.setForeground (Color.yellow);
    pb.setMaximum (1000);
    pb.setValue (50);
    pb.setBorder (
        BorderFactory.createRaisedBevelBorder ());
}
if (i == 2) {
    pb.setForeground (Color.blue);
    pb.setBorderPainted (false);
    pb.setValue (50);
    pb.setStringPainted (true);
}
if (i == 3) {
    pb.setOrientation (JProgressBar.VERTICAL);
    pb.setForeground (Color.red);
    pb.setValue (90);
    pb.setStringPainted (true);
    pb.setBorder (
        BorderFactory.createEtchedBorder ());
}
contentPane.add (pb);
}
}

```

这个小应用程序创建了一组进度条，并为每个进度条设置了各种不同的属性，从而产生了图 11-1 中示出的各种不同配置。

11.1.1 进度条与线程

进度条主要用于反映一个耗时任务已完成的时间比例或剩余时间比例。但是，耗时任务不应当从事件派发线程中完成，而 Swing 组件应当只从事件批发线程中更新。那么，如何在一个独立线程的基础上更新一个进度条呢？正如 2.4 节“Swing 和线程”中讨论的那样，可以通过用 `SwingUtilities.invokeLater` 方法使一个独立线程更新一个进度条来解决这个矛盾。

图 11-2 中示出的小应用程序递增一个进度条的值，每秒递增一次，从进度条的最小值递增到它的最大值。激活这个小应用程序中的按钮就开始了这个递增过程。图 11-2 中的左图示出了这个小应用程序在启动按钮刚被激活后的样子。右图则示出了这个小应用程序正在更新进度条的样子。

这个小应用程序创建了一个 `JProgressBar` 实例和一个 `JButton` 实例，它们都添加到这个小应用程序的内容窗格中。在启动按钮上还添加了一个监听器，该监听器创建了一个随后启动的 `UpdateThread` 实例。

```

public class Test extends JApplet {
    private JProgressBar progressBar = new JProgressBar ();
    private JButton startButton = new JButton ("start");

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (startButton);
        contentPane.add (progressBar);

        progressBar.setStringPainted (true);

        startButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                (new UpdateThread ()) .start ();
            }
        });
    }
}
...

```

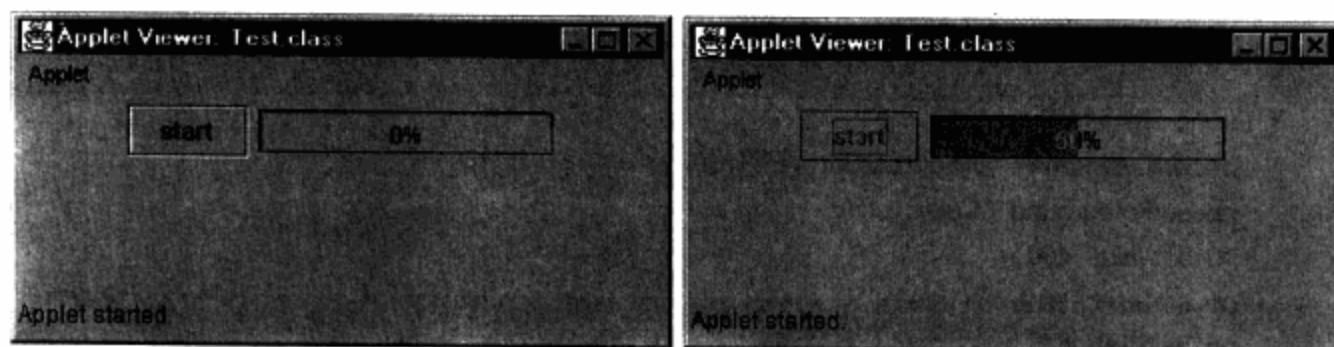


图 11-2 一个耗时任务的进度条

UpdateThread 类是这个小程序的内部类，它扩展 `java.lang.Thread`。UpdateThread 的构造方法创建两个 Runnable 实例：update 递增与进度条的值，finish 则把进度条的值复位为它的最小值。

```

...
class UpdateThread extends Thread {
    Runnable update, finish;
    int value, min, max, increment;

    public UpdateThread () {
        max = progressBar.getMaximum ();
        min = progressBar.getMinimum ();

        update = new Runnable () {
            public void run () {
                value = progressBar.getValue () + increment;
                updateProgressBar (value);
            }
        };
        finish = new Runnable () {
            public void run () {
                updateProgressBar (min);
            }
        };
    }
}
...

```

UpdateThread 类的 run 方法通过休眠一秒来仿真一个耗时的活动。simulateTime-

ConsumingActivity 方法设置滑杆的递增量，然后调用 SwingUtilities.invokeLater ()，调用参数是 update Runnable。当达到这个进度条的最大值时，再次调用 SwingUtilities.invokeLater ()，这次的调用参数是 finish Runnable。注意，当其值更新时重绘进度条是没有必要的，因为调用 JProgressBar.setValue () 导致进度条重绘一次。

```
...
public void run () {
    startButton.setEnabled (false);

    while (value + increment <= max) {
        simulateTimeConsumingActivity ();
        SwingUtilities.invokeLater (update);
    }

    SwingUtilities.invokeLater (finish);
    startButton.setEnabled (true);
}
...
}
```

例 11-2 中完整地列出了图 11-2 中示出的小应用程序的代码。

例 11-2 使用 JProgressBar

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private JProgressBar progressBar = new JProgressBar ();
    private JButton startButton = new JButton ("start");

    public void init () {
        Container contentPane = getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (startButton);
        contentPane.add (progressBar);

        progressBar.setStringPainted (true);

        startButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                (new UpdateThread ()) .start ();
            }
        });
    }

    class UpdateThread extends Thread {
        Runnable update, finish;
        int value, min, max, increment;

        public UpdateThread () {
            max = progressBar.getMaximum ();
            min = progressBar.getMinimum ();

            update = new Runnable () {
                public void run () {
                    value = progressBar.getValue () + increment;
                    updateprogressBar (value);
                }
            };
            finish = new Runnable () {
                public void run () {
                    progressBar.setValue (max);
                }
            };
        }

        public void run () {
            update.run ();
            finish.run ();
        }
    }
}
```


绘制器： ——
编辑器： ——
激发的事件： ChangeEvent
替代： ——
类图：

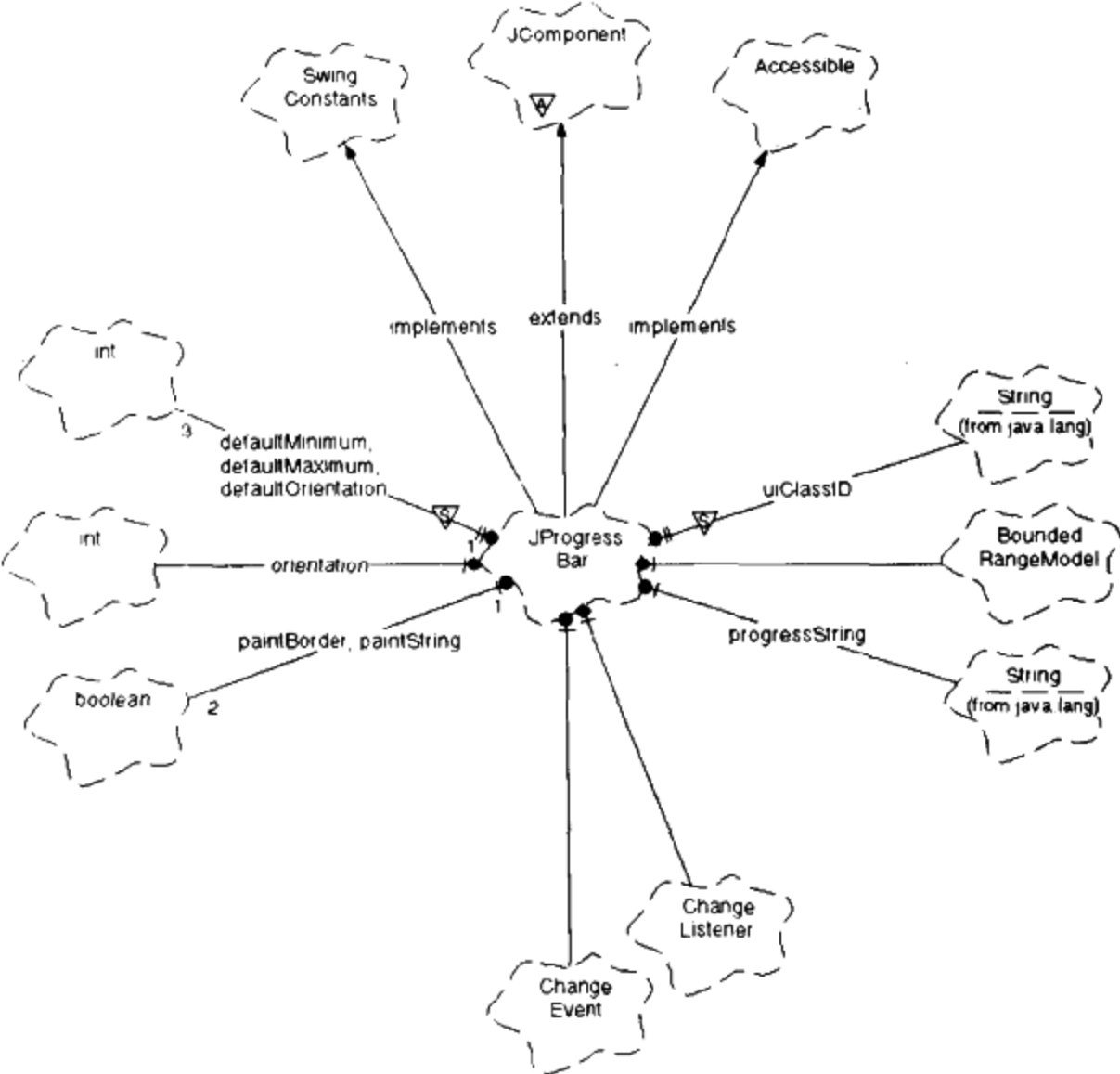


图 11-3 JProgressBar 类图

JProgressBar 扩展 JComponent，实现 SwingConstants 和 Accessible 接口，并维护对其模型（一个 BoundedRangeModel 实例的 protected 引用）。JProgressBar 还维护对一些 integer 和 boolean 变量的引用，这些引用分别记录进度条的方向及是否绘制一个边框或进度字符串。

与所有激发变化事件的 Swing 组件一样，JProgressBar 类只使用一个 ChangeEvent 实例向它的变化监听器激发变化事件，与变化事件有关的唯一状态是事件源。有关变化事件参见 3.2.4 节。

11.1.2 JProgressBar 属性

表 11-1 中列出了 JProgressBar 维护的属性。

表 11-1 JProgressBar 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
borderPainted	boolean	B	SG	true ^④
maximum	int	Ch	CSG	100
minimum	int	Ch	CSG	0
model	Bounded RangeModel	Ch	SG	DefaultBounded RangeModel

(续)

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
orientation	int	B	CSG	HORIZONTAL
percentComplete	double	B	G	——
string	String	B	SG	false
stringPainted	boolean	B	SG	false
value	int	Ch	SG	0

- ① B = 关联的 (激发 PropertyChangeEvent) / C = 受约束的 / I = 索引的 / S = 简单的 / Ch = 激发 ChangeEvent
- ② G = 可在创建时设置 / G = 获取方法 / S = 设置方法
- ③ I&F = 与界面样式有关
- ④ 缺省边框是 lowered bevel border???

borderPainted—— 如果值为 true，则在进度条外围绘制一个边框。

maximum—— 进度条表示的最大值。

minimum—— 进度条表示的最小值。

model—— BoundedRangeModel 接口的一个实现；缺省时，一个进度条的模型是 BoundedRangeModel 的一个实例。

orientation—— JProgressBar.HORIZONTAL 或 JProgressBar.VERTICAL。

percentComplete—— 一个进度条的模型维护最小值、最大值和当前值。这个属性是用以下方式计算的：

$$(value - minimum) / (maximum - minimum)$$

string—— 进度条显示的字符串。缺省时，从 JProgressBar.getString () 返回的字符串是已完成的百分比，其后跟有一个 “%” 符。这个字符串可以显式地设置。在这种情况下，这个字符串通常用 percentComplete 属性创建。

stringPainted—— 一个 boolean 属性，决定是否显示一个进度条字符串。缺省时，stringPainted 属性为假。

value—— 进度条的滑杆表示的值。

对任何 JProgressBar 属性的修改都将导致重绘进度条一次。另外，对 JProgress 属性的修改导致激发变化事件或属性变化事件。

11.1.3 JProgressBar 事件

JProgressBar 实例的 value、minimum、maximum 和 model 属性的变化将导致激发变化事件。图 11-4 中示出的小应用程序与图 11-2 中示出的小应用程序基本相同，不同之处是在它的进度条上添加了一个变化监听器。当进度条激发一个变化事件时，它的最小值、最大值和当前值显示在该小应用程序的状态区中。

变化监听器直接从组件获得进度条的最小值、最大值和当前值，因为变化事件仅记录事件源。

```
public class Test extends JApplet {
    private JProgressBar pb = new JProgressBar ();
    public void init () {
        ...
        pb.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                int min = pb.getMinimum (), max = pb.getMaximum ();
```

```

        int value = pb.getValue ();
        showStatus ("Min: " + min + ", Max: " + max +
            ", Value: " + value);
    }
}

```

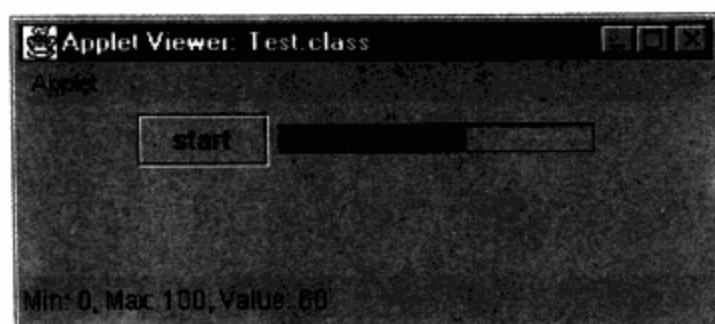


图 11-4 监视一个进度条的值

例 11-3 中列出了图 11-4 中示出的小应用程序的代码。

例 11-3 监视一个进度条的值

```

import javax.swing. * ;
import javax.swing.event. * ;
import java.awt. * ;
import java.awt.event. * ;

public class Test extends JApplet {
    private JProgressBar pb = new JProgressBar ();

    public void init () {
        Container contentPane = getContentPane ();
        final JButton startButton = new JButton ("start");

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (startButton);
        contentPane.add (pb);

        startButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                (new UpdateThread (pb)).start ();
            }
        });

        pb.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                int min = pb.getMinimum (), max = pb.getMaximum ();
                int value = pb.getValue ();

                showStatus ("Min: " + min + ", Max: " + max +
                    ", Value: " + value);
            }
        });
    }

    class UpdateThread extends Thread {
        Runnable update, finish;
        JProgressBar pb;
    }
}

```

```

int value, min, max, increment;

public UpdateThread (JProgressBar progressBar) {
    pb = progressBar;

    max = pb.getMaximum ();
    min = pb.getMinimum ();

    update = new Runnable () {
        public void run () {
            value = pb.getValue () + increment;
            pb.setValue (value);
        }
    };

    finish = new Runnable () {
        public void run () {
            value = min;
            pb.setValue (value);
        }
    };

    public void run () {
        while (value + increment <= max) {
            simulateTimeConsumingActivity ();
            SwingUtilities.invokeLater (update);
        }
        SwingUtilities.invokeLater (finish);
    }

    private void simulateTimeConsumingActivity () {
        try {
            Thread.currentThread ().sleep (1000);
            increment = (max - min) / 10;
        }
        catch (InterruptedException e) {
            e.printStackTrace ();
        }
    }
}

```

11.1.4 JProgressBar 类总结

类总结 11-1 中列出了 JProgressBar 类的 public 和 protected 变量和方法。

类总结 11-1 JProgressBar

1. 构造方法

```

public JProgressBar ()
public JProgressBar (int orientation)
public JProgressBar (int minimum, int maximum)
public JProgressBar (int orientation, int minimum, int maximum)
public JProgressBar (BoundedRangeModel)

```

JProgressBar 的无参数构造方法用表 11-1 中列出的缺省属性值创建进度条。方向、最小值、最大值和进度条的模型均在创建时刻指定。

2. 方法

(1) 状态变化

```

public void addChangeListener (ChangeListener)
public void removeChangeListener (ChangeListener)

protected ChangeListener createChangeListener ()
protected void fireStateChanged ()

```

与大多数 Swing 组件一样，当修改与其模型有关的属性时，JProgressBar 激发变化事件。JProgressBar 为在其实例上添加和删除变化监听器提供了 public 方法。CreateChangeListener 方法创建一个 JProgressBar.ModelListener 实例来处理进度条的模型激发的变化事件，并把它们传递给组件的变化监听器。fireStateChanged 方法向进度条的变化监听器激发变化事件。

(2) 进度条字符串

```

public String getString ()
public void setString ()

public boolean isStringPainted (boolean)
public void setStringPainted (boolean)

```

上面列出的方法是 string 和 stringPainted 属性的访问方法。有关这些属性的更多信息参见表 11-1。

(3) 更新

```

public void update (Graphics)
protected void paintBorder (Graphics)

```

通常，Swing 和 AWT 组件在它们的 update 方法中清除并重画它们的背景。JProgressBar 重载了 update () 方法，它直接调用 paint，从而放弃了清除背景的操作。

paintBorder 方法重载 JComponent 中的 paintBorder 方法，仅当 borderPainted 属性为 true 时才绘制边框。

(4) 杂项属性

```

public double getPercentComplete ()
public int getMaximum ()
public int getMinimum ()
public BoundedRangeModel getModel ()
public int getOrientation ()
public int getValue ()

public void setBorderPainted (boolean)
public void setMaximum (int)
public void setMinimum (int)
public void setModel (BoundedRangeModel)
public void setOrientation (int)
public void setString (String)
public void setStringPainted (boolean)
public void setValue (int)

public boolean isBorderPainted ()

```

上面列出的方法是 JProgressBar 属性的访问方法。由于 JProgressBar 只提供了一个无参数的构造方法，所以它为所有属性均提供了设置方法和获取方法。

(5) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public ProgressBarUI getUI ()
public String getUIClassID ()
public void setUI (ProgressBarUI)
public void updateUI ()

```

上面方法可以在 JComponent 的大多数扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

11.1.5 AWT 兼容

AWT 没有提供与 JProgressBar 类似的组件。

11.2 JSlider

与 JProgressBar 一样, JSlider 显示一个介于最小值与最大值之间的值。进度条显示的值只能由程序操纵,而滑杆的值能够直接拖动滑杆柄或单击滑杆槽来操纵。Swing 滑杆包含一个可推动的柄,还可以带主要的和次要的间隔标记及标签。

图 11-5 中示出的小应用程序包含一个 Metal 界面样式的 JSlider 实例。

例 11-4 中列出了图 11-5 中示出的小应用程序的代码。

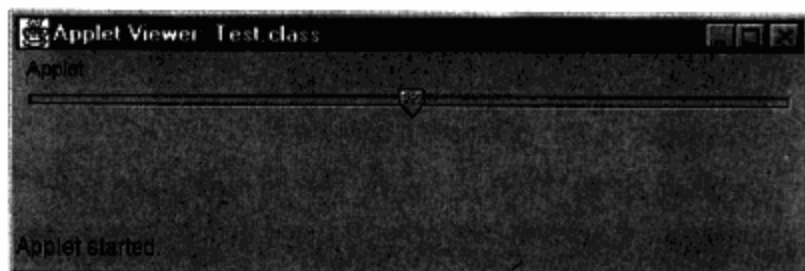


图 11-5 JSlider 的简单例

例 11-4 一个简单的 JSlider 例子

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane () ;
        JSlider slider = new JSlider () ;
        contentPane.add (slider, BorderLayout.NORTH);
    }
}
```

这个小应用程序用一个无参数构造方法创建了一个 JSlider 实例,这个实例的最小值、最大值和初始值分别为 0、100 和 50。在创建这个滑杆后,把它加入到这个小应用程序的内容窗格中。

11.2.1 填充的滑杆

如果一个滑杆的界面样式是 Java 界面样式,则它就能与图 11-6 中示出的那样,通过设置一个客户属性 “JSlider.isFilled” 来填充滑槽。

例 11-5 中列出了图 11-6 中示出的小应用程序的代码。

例 11-5 一个填充的滑杆

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
```



```

public Test () {
    Container contentPane = getContentPane ();
    JSlider slider = new JSlider ();

    slider.putClientProperty ("JSlider.isFilled", Boolean.TRUE);
    contentPane.add (slider, BorderLayout.NORTH);
}

```

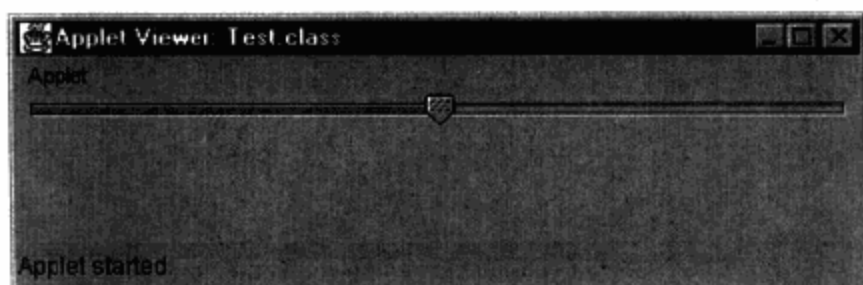


图 11-6 一个填充的滑杆

用 true 值设置客户属性 “JSlider.isFilled”，这样滑柄左边的滑槽就填充了。如果把这个属性设置为 false 或根本就不设置这个属性，则滑槽不被填充。

11.2.2 滑杆间隔标记

滑杆能够绘制间隔标记，间隔标记是用来描述与滑杆有关的特定值的。间隔标记有主要标记和次要标记之别。主要间隔标记表示一些特定的值，而次要标记表示介于主要标记之间的值。主要和次要间隔标记的外观取决于滑杆的界面样式，但主要间隔标记一般比次要间隔标记大。

有三个方法与绘制滑杆的间隔标记有关：setPaintTicks ()、setMinorTickSpacing () 和 setMajorTickSpacing ()。setPaintTicks () 带一个 boolean 变量，指示是否应该绘制间隔标记。setMinorTickSpacing () 和 setMajorTickSpacing () 方法带一些 integer 值，这些值分别表示主要标记和次要标记之间的单元数量。关于这些方法有两点需要指出：

首先，用一个 true 值调用 setPaintTicks () 导致仅当设置了主要或次要间隔标记之间的间隙时才绘制间隔标记。换言之，如果不用一个对于 0 的值调用 setMinorTickSpacing () 或 setMajorTickSpacing ()，则调用 setPaintTicks (true) 将不会绘制间隔标记。

其次，传送给 setMinorTickSpacing () 和 setMajorTickSpacing () 方法的 integer 值不直接表示间隔标记之间的像素点数。这些 integer 值表示的是间隔标记之间的单元数。例如，调用 setMajorTickSpacing (25) 指定主要间隔标记之间应该相隔 25 个单元。如果一个滑杆的最小值和最大值分别是 0 和 100，则指定主要间隔标记为 25 将产生 5 个主要间隔标记——分别在 0、25、50、75 和 100 处绘制。如果改变了这个滑杆的大小，这 5 个主要标记将重新绘制。

图 11-7 中示出的小应用程序包含一个 JSlider 实例和一个复选框，以及两个用于设置间隔绘制属性和主要间隔标记与次要间隔标记之间的间隙的组合框。这个小应用程序是以 Windows 界面样式显示的，因为这样比用 Metal 界面样式更易于区分间隔标记。

图 11-7 中最上面的图示出了这个小应用程序最初的样子。

第二幅图示出的是选取 “Paint Ticks” 复选框后这个小应用程序的样子。注意，其中未显示任何间隔标记，因为还没有设置主要或次要间隔标记之间的间隙。

第三幅图示出了设置主要间隔标记之间的间隙后这个小应用程序的样子，其中显示了主要

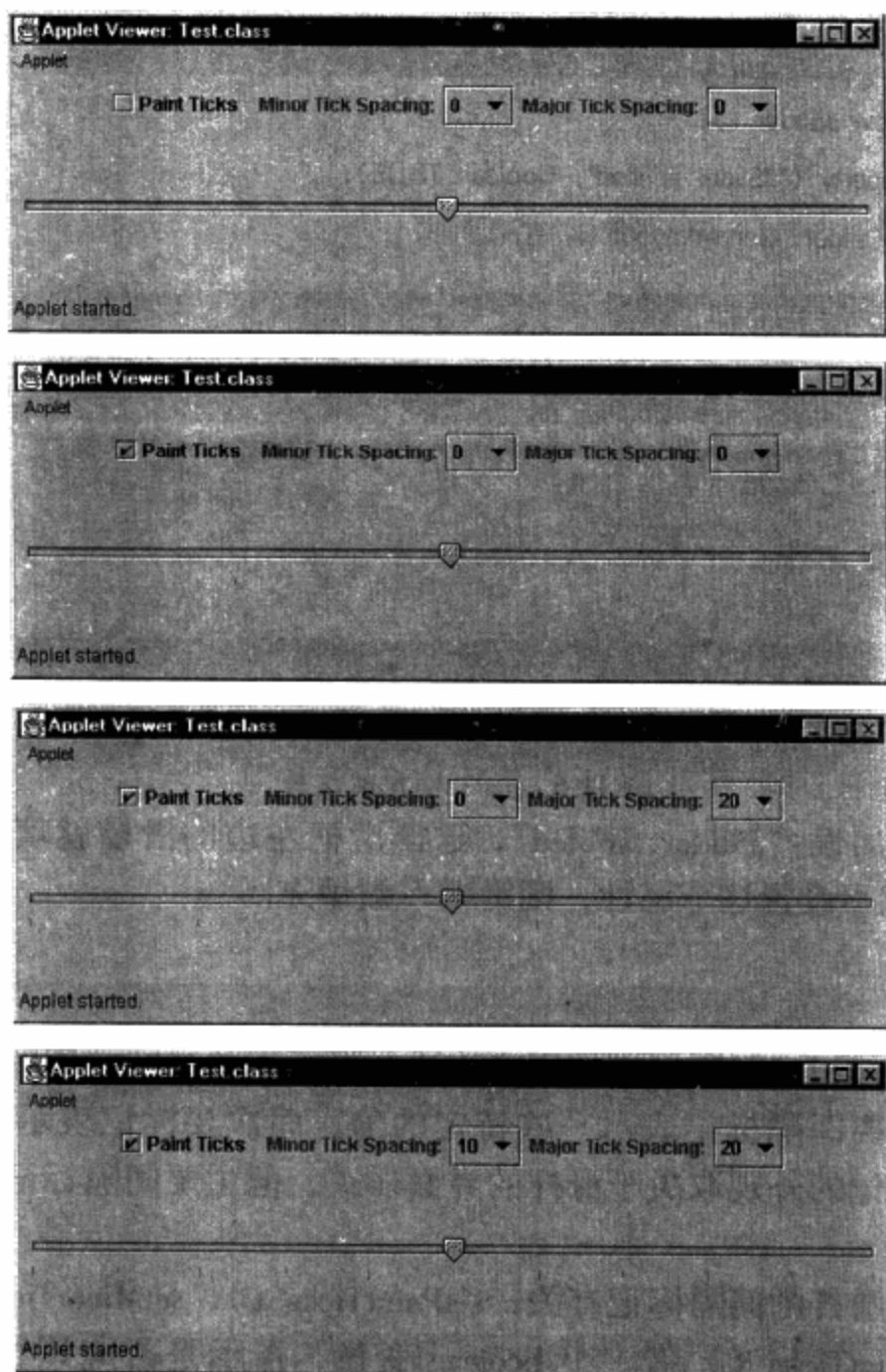


图 11-7 滑杆的间隔标记

间隔标记。

最后一幅图显示了再设置次要间隔标记之间的间隙后这个小应用程序的样子，其中同时显示了主要间隔标记和次要间隔标记。

这个小应用程序用 `JSlider` 的无参数构造方法创建了一个 `JSlider` 实例。这个小应用程序还创建了一个 `ControlPanel` 实例。`ControlPanel` 是 `JPanel` 的一个扩展，用于放置复选框和组合框。把这个控制面板设计为北组件添加到这个小应用程序中，而指定滑杆为内容窗格的中心组件。

```
public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JSlider slider = new JSlider ();
        JPanel controlPanel = new ControlPanel (slider);
        contentPane.add (controlPanel, BorderLayout.NORTH);
        contentPane.add (slider, BorderLayout.CENTER);
    }
}
```

复选框和组合框上添加了操纵滑杆的监听器。当“Paint Ticks”激活复选框时，调用 `JSlider.setPaintTicks ()`。调用这个方法时带了一个表示复选框选取状态的 `boolean` 值。当在组合框中

进行了选取时，则用从复选框选取的值调用 `setMinorTickSpacing()` 或 `setMajorTickSpacing()`。

正如表 11-2 中所示，任何由复选框或组合框操纵的属性都不会导致滑杆外观的自动更新。因此，对滑杆调用了 `revalidate()` 方法，以便重新绘制滑杆。有关 `revalidate()` 方法的详细内容参见 4.3.5 节“`validate`、`invalidate` 和 `revalidate` 方法”。

```
class ControlPanel extends JPanel {
    public ControlPanel (final JSlider slider) {
        JCheckBox paintTicks = new JCheckBox ("Paint Ticks");
        JComboBox minorSpacing = new JComboBox (),
            majorSpacing = new JComboBox ();

        ...

        paintTicks.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JCheckBox cb = (JCheckBox) e.getSource ();
                slider.setPaintTicks (cb.isSelected ());
                slider.revalidate ();
            }
        });

        minorSpacing.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                JComboBox cb = (JComboBox) e.getSource ();
                int spacing = Integer.parseInt (
                    (String) cb.getSelectedItem ());

                slider.setMinorTickSpacing (spacing);
                slider.revalidate ();
            }
        });

        majorSpacing.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                JComboBox cb = (JComboBox) e.getSource ();
                int spacing = Integer.parseInt (
                    (String) cb.getSelectedItem ());

                slider.setMajorTickSpacing (spacing);
                slider.revalidate ();
            }
        });
    }
}
```

例 11-6 列出了图 11-7 中示出的小应用程序的代码。

例 11-6 显示滑杆的间隔标记

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public test () {
        Container contentPane = getContentPane ();
        JSlider slider = new JSlider ();
        JPanel controlPanel = new ControlPanel (slider);
        contentPane.add (controlPanel, BorderLayout.NORTH);
        contentPane.add (slider, BorderLayout.CENTER);
    }
}
```

```
    }  
}  
class ControlPanel extends JPanel {  
    public ControlPanel (final JSlider slider) {  
        JCheckBox paintTicks = new JCheckBox ("Paint Ticks");  
        JComboBox minorSpacing = new JComboBox (),  
            majorSpacing = new JComboBox ();  
  
        minorSpacing.addItem ("0");  
        minorSpacing.addItem ("3");  
        minorSpacing.addItem ("5");  
        minorSpacing.addItem ("10");  
        minorSpacing.addItem ("20");  
  
        majorSpacing.addItem ("0");  
        majorSpacing.addItem ("3");  
        majorSpacing.addItem ("5");  
        majorSpacing.addItem ("10");  
        majorSpacing.addItem ("20");  
  
        add (paintTicks);  
        add (new JLabel ("Minor Tick Spacing:"));  
        add (minorSpacing);  
        add (new JLabel ("Major Tick Spacing:"));  
        add (majorSpacing);  
  
        paintTicks.addActionListener (new ActionListener () {  
            public void actionPerformed (ActionEvent e) {  
                JCheckBox cb = (JCheckBox) e.getSource ();  
                slider.setPaintTicks (cb.isSelected ());  
                slider.repaint ();  
            }  
        });  
  
        minorSpacing.addItemListener (new ItemListener () {  
            public void itemStateChanged (ItemEvent e) {  
                JComboBox cb = (JComboBox) e.getSource ();  
                int spacing = Integer.parseInt (  
                    (String) cb.getSelectedItem ());  
  
                slider.setMinorTickSpacing (spacing);  
                slider.revalidate ();  
            }  
        });  
  
        majorSpacing.addItemListener (new ItemListener () {  
            public void itemStateChanged (ItemEvent e) {  
                JComboBox cb = (JComboBox) e.getSource ();  
                int spacing = Integer.parseInt (  
                    (String) cb.getSelectedItem ());  
  
                slider.setMajorTickSpacing (spacing);  
                slider.revalidate ();  
            }  
        });  
    }  
}
```

11.2.3 滑杆标签

除了显示间隔标记外, JSlider 实例还能显示滑杆标签。滑杆标签是在滑杆的主要间隔标记位置上绘制的标签。JSlider 能够自己产生数字标签, 还能通过提供一个 Hashtable 来指定定制标签。Hashtable 中包含表示值和相应标签的 Integer/JLabel 参数对。

图 11-8 中示出的小应用程序包含一个显示标准 (即由 JSlider 类产生的) 标签的 JSlider 实例。

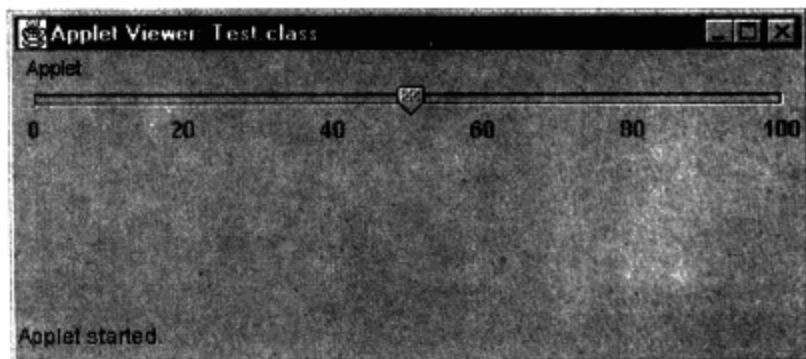


图 11-8 缺省的 JSlider 标签

例 11-7 列出了图 11-8 中示出的小应用程序的代码。

例 11-7 显示缺省的 JSlider 标签

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JSlider slider = new JSlider ();
        slider.setPaintLabels (true);
        slider.setMajorTickSpacing (20);
        contentPane.add (slider, BorderLayout.NORTH);
    }
}
```

这个小应用程序调用了 JSlider.setPaintLabels (true), 指示应该为滑杆绘制标签。但是, 为了绘制标签仅调用 setPaintLabels () 是不够的。在为主要间隔设置间隙之前, JSlider 类不能确定应当在何处绘制标签。因此, 为了显示标签, 必须调用 setMajorTickSpacing () 方法。

Swing 提示

仅当指定了间隔标记之间的间隙后才绘制间隔标记和标签

JSlider 类提供了分别使标签和间隔标记可视化的方法: setPaintLabels () 和 setPaintTicks ()。这两个方法都带一个布尔值作为参数——值为真使标签和间隔标记可见, 值为假则使它们不可见。但是, 调用 setPaintLabels (true) 或 setPaintTicks (true) 并不能使标签和间隔标记一定可见。

除非显式地设置了间隔标记之间的间隙, 否则不会绘制自动产生的标签和间隔标记。因为间隔间隙不缺省为一个合适的值这是与 Swing 组件的大多数属性不同的。在绘制自动产生的标签和间隔标记能够之前, 必须显式地设置间隔标记之间的间隙[⊖]。

⊖ 显示定制标签不必显式地设置间隔标记之间的间隙。

定制的滑杆标签

除标准标签外，还能为 JSlider 实例定制标签。如图 11-9 所示。



图 11-9 定制的滑杆标签

例 11-8 列出了图 11-9 中示出的小应用程序的代码。

例 11-8 定制滑杆标签

```
import java.util. * ;
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JSlider slider = new JSlider ();

        Icon [] icons = {
            new ImageIcon ("basketball.gif"),
            new ImageIcon ("baseball.gif"),
            new ImageIcon ("soccer.gif"),
            new ImageIcon ("clipboard.gif"),
            new ImageIcon ("filmstrip.gif"),
            new ImageIcon ("crab.gif"),
        };

        Hashtable table = new Hashtable ();
        for (int i=0, loc=0; i < icons.length; i++, loc += 20) {
            table.put (new Integer (loc),
                new JLabel (Integer.toString (loc),
                    icons [i],
                    JLabel.LEFT));
        }

        slider.setLabelTable (table);
        slider.setPaintLabels (true);
        slider.setMajorTickSpacing (20);
        contentPane.add (slider, BorderLayout.NORTH);
    }
}
```

这个小应用程序调用了 JSlider.setLabelTable () 方法，其调用参数是一个包含 Integer/Component 值对的 Hashtable，这里的组件是标签。JLabel 的每个实例都是用一个图标和表示一个值的文本创建的。这个小应用程序的 JSlider UI 代表用这个哈希表来绘制标签和相应的文本。

随后还调用了 `setPaintLabels()` 和 `setMajorTickSpacing()` 方法，对绘制标签来说，这两个方法都是必须的。

11.2.4 反转滑杆值

所有的滑杆都维护一个 `inverted` 属性，这个属性决定滑杆值的递增方向。缺省情况下，`inverted` 属性的值为 `false`；并且，水平滑杆的值从左向右递增，垂直滑杆的值从下向上递增。当 `inverted` 属性的值为 `true` 时，水平滑杆的值从右向左递增，垂直滑杆的值则从上向下递增。

从图 11-10 中示出的小应用程序中可以看到设置 `inverted` 属性的效果。图 11-10 中左图显示了滑杆标签的缺省方向，而右图显示了反转后的标签。

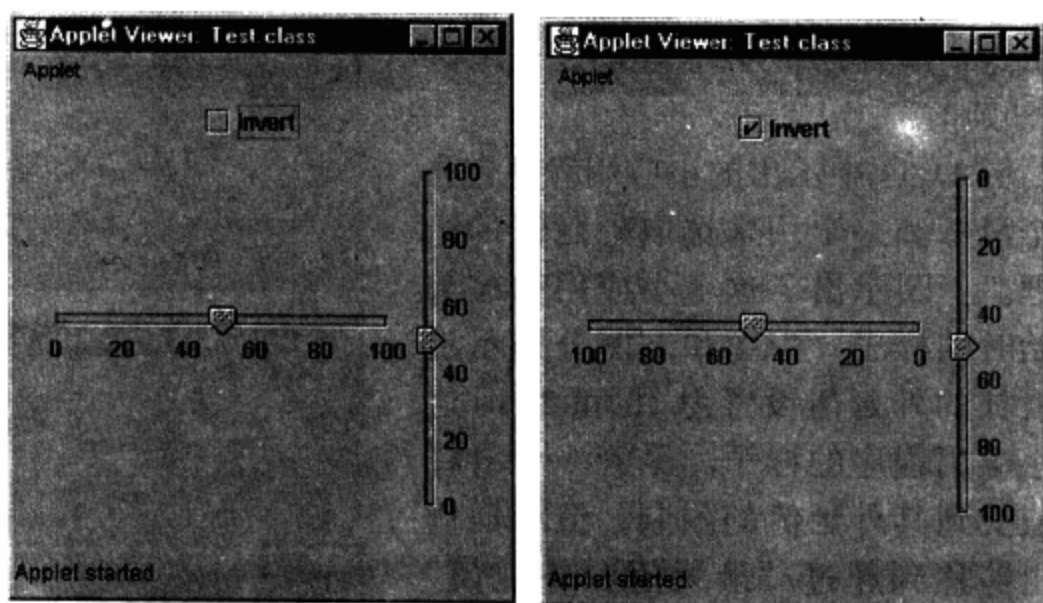


图 11-10 反转滑杆

例 11-9 中列出了图 11-10 中示出的小应用程序的代码

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JCheckBox checkBox = new JCheckBox("Invert");
    private JSlider[] sliders = { new JSlider(),
                                   new JSlider(JSlider.VERTICAL) };

    public Test() {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        contentPane.add(checkBox);

        for (int i = 0; i < sliders.length; ++i) {
            sliders[i].setPaintLabels(true);
            sliders[i].setMajorTickSpacing(20);
            contentPane.add(sliders[i]);
        }

        checkBox.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for (int i = 0; i < sliders.length; ++i)
                    sliders[i].setInverted(checkBox.isSelected());
            }
        });
    }
}
```


两个滑杆都分别把它们的 PaintLabels 属性和 MajorTickSpacing 属性设置为 true 和 20，并且这两个滑杆都添加到这个小应用程序的内容窗格中。

当激活复选框时，两个滑杆根据是否选取复选框来设置它们的反转属性。

11.2.5 滑杆的外延值

滑杆的外延确定滑杆值的较高范围。例如，如果一个滑杆的最大值为 100，外延值为 20，那么这个滑杆的值将不会超过“最大值-外延值”。

图 11-11 中示出的小应用程序包含一个滑杆和一个用来选取外延值的组合框。上图示出了这个小应用程序最开始时的样子，中图和下图分别示出了滑杆的外延值设置为 10 和 20 时这个小应用程序的样子。尽管一个滑杆的值受到其外延值的限制，但滑柄仍然能够移到超过“最大值和外延值”的位置上^①。

这个小应用程序创建了一个 JSlider 实例和一个包含组合框的面板。在这个组合框上添加了一个子项监听器，这个监听器根据组合框中的选取值设置这个滑杆的外延值。

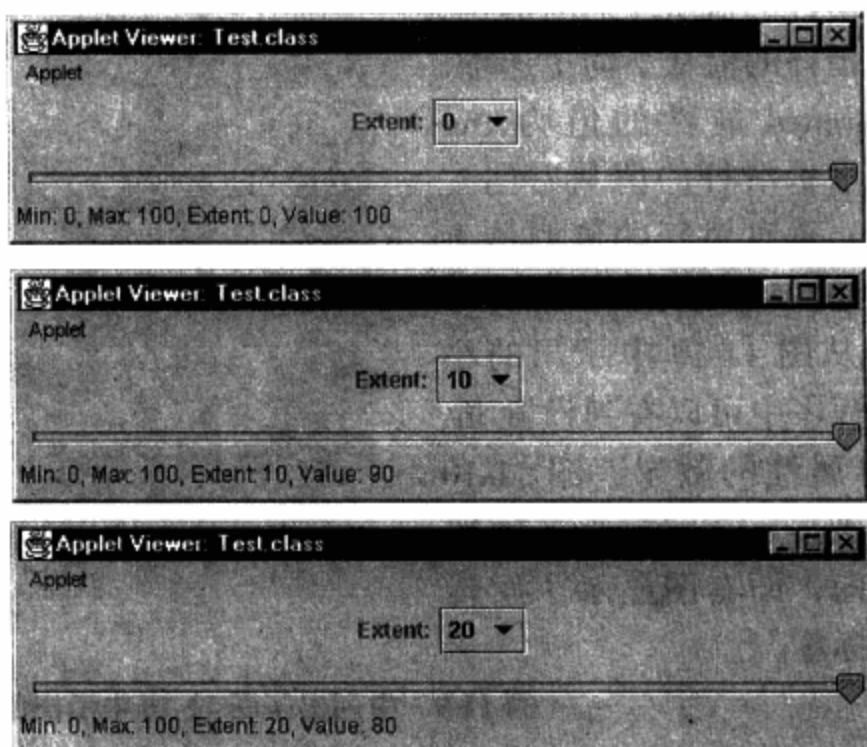


图 11-11 滑杆的外延

```
class ControlPanel extends JPanel {
    public ControlPanel (final JSlider slider) {
        JComboBox extent = new JComboBox ();

        extent.addItem ("0");
        extent.addItem ("10");
        extent.addItem ("20");
        extent.addItem ("30");
        extent.addItem ("40");

        add (new JLabel ("Extent:"));
        add (extent);

        extent.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                JComboBox cb = (JComboBox) e.getSource ();
                int ext = Integer.parseInt (
                    (String) cb.getSelectedItem ());

                slider.setExtent (ext);
            }
        });
    }
}
```

① Swing 的早期版本可把滑柄移到“最大值-外延值”的位置上。

例 11-10 完整地列出了图 11-11 中示出的小应用程序的代码。

例 11-10 设置一个滑杆的外延值

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JSlider slider = new JSlider ();
        JPanel controlPanel = new ControlPanel (slider);

        contentPane.add (controlPanel, BorderLayout.NORTH);
        contentPane.add (slider, BorderLayout.SOUTH_CENTER);

        slider.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                JSlider s = (JSlider) e.getSource ();
                showStatus ("Min: " + s.getMinimum () +
                    ", Max: " + s.getMaximum () +
                    ", Extent: " + s.getExtent () +
                    ", Value: " + s.getValue ());
            }
        });
    }
}

class ControlPanel extends JPanel {
    public ControlPanel (final JSlider slider) {
        JComboBox extent = new JComboBox ();

        extent.addItem ("0");
        extent.addItem ("10");
        extent.addItem ("20");
        extent.addItem ("30");
        extent.addItem ("40");

        add (new JLabel ("Extent:"));
        add (extent);

        extent.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                JComboBox cb = (JComboBox) e.getSource ();
                int ext = Integer.parseInt (
                    (String) cb.getSelectedItem ());

                slider.setExtent (ext);
            }
        });
    }
}
```

组件总结 11-2 对 JSlider 组件进行了总结。

组件总结 11-2 JSlider

模型: BoundedRangeModel

UI 代表: javax.swing.plaf.basic.BasicSliderUI
绘制器: ——
编辑器: ——
激发的事件: ——
替代: ——
类图:

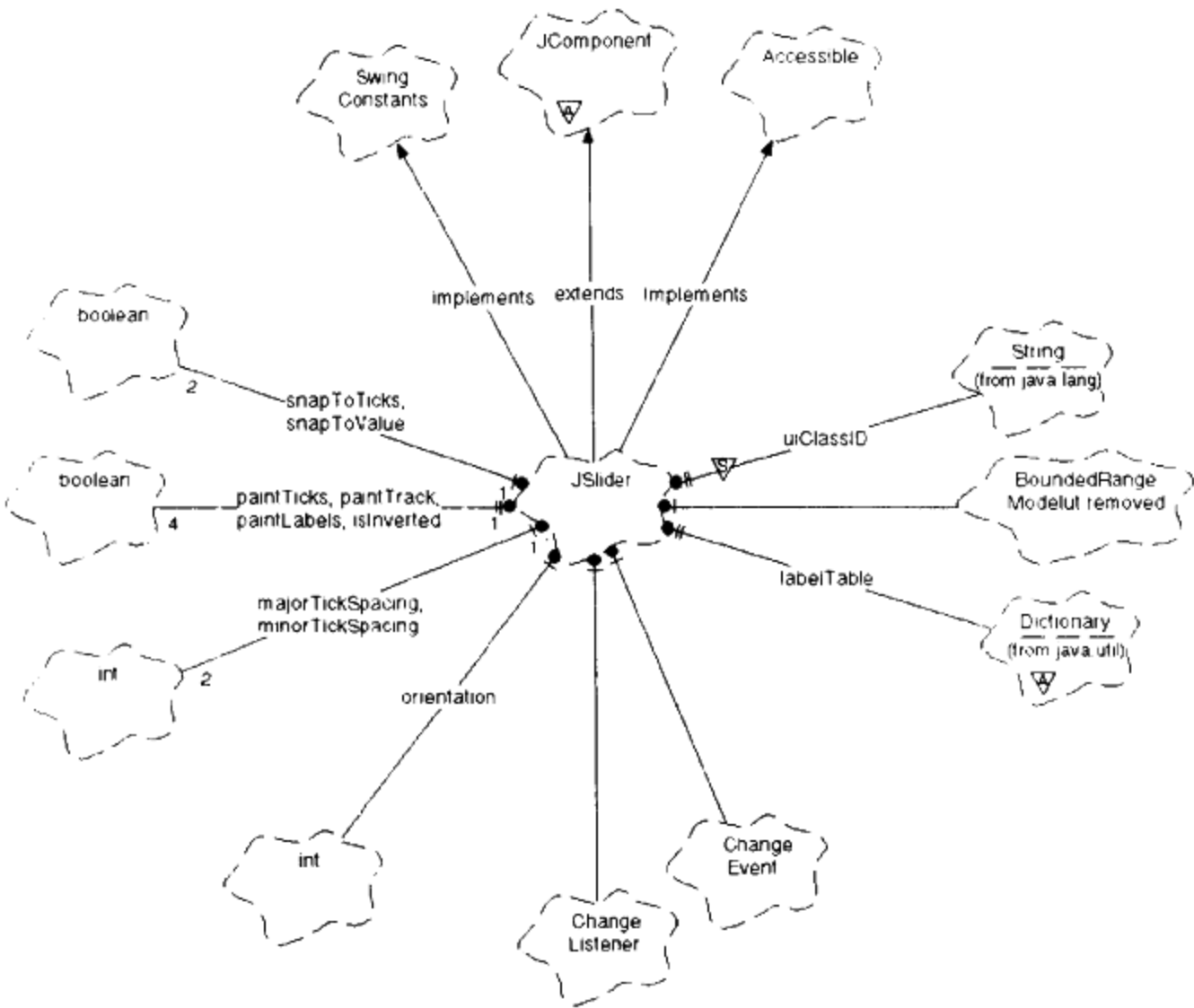


图 11-12 JSlider 的类图

JSlider 的模型是一个 BoundedRangeModel 实例，并且 JSlider 类维护对其模型的一个引用。JSlider 还维护对一个 Dictionary 的引用，Dictionary 中包含了一些 Integer/Component 值对，这些值对表示滑杆的值和在滑杆值的位置上显示的相应组件。

JSlider 维护 6 个属性的 protected 布尔引用，这些属性影响间隔标记、滑杆滑轨和滑杆标签的可视性。snapToTicks 和 snapToValue 布尔变量决定滑柄对应一个间隔标记还是对应一个值。

与所有激发变化事件的 Swing 组件一样，JSlider 类只使用一个 ChangeEvent 实例向它的变化监听器激发变化事件，因为与一个变化事件有关的唯一状态是事件源。

一个滑杆的模型，即一个 BoundedRangeModel 实例在修改模型的值时激发变化事件。JSlider 实例向它们的模型登记变化监听器并把变化事件传递给它们自己的变化监听器；因此 JSlider 类维护对 ChangeEvent 和 ChangeListener 实例的 protected 引用。

11.2.6 JSlider 属性

表 11-2 中列出了 JSlider 类维护的属性。

表 11-2 JSlider 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
extent	int	Ch	SG	0
inverted	boolean	B	SG	false
labelTable	Directory	B	SG	——
majorTickSpacing	int	B	SG	——
maximum	int	Ch	CSG	100
minimum	int	Ch	CSG	0
minorTickSpacing	int	B	SG	——
model	Bounded RangeModel	B	CSG	DefaultBounded RangeModel
orientation	int	B	CSG	HORIZONTAL
paintLabels	boolean	B	SG	false
paintTicks	boolean	B	SG	false
snapToTicks	boolean	Ch	SG	false
value	int	Ch	CSG	50
valuesIsAdjusting	boolean	Ch	SG	false

① B = 关联的（激发 PropertyChangeEvent）/C = 受约束的/ I = 索引的/
S = 简单的/Ch = 激发 ChangeEvent

② C = 可在创建时设置/G = 获取方法/S = 设置方法

③ L&F = 与界面样式有关

- extent** —— 定义滑杆值范围的较高部分。
- inverted** —— 如果值为 true，则值从右向左或从下向上递增。
- labelTable** —— 标签/值对的 Dictionary。
- majorTickSpacing** —— 主要间隔标记间的单元数。
- maximum** —— 滑杆表示的最大值。
- minimum** —— 滑杆表示的最小值。
- minorTickSpacing** —— 次要间隔标记间的单元数。
- model** —— JSlider 实例的模型，JSlider 是 BoundedRangeModel 接口的一个实现。
- orientation** —— JSlider 的方向，JSlider.HORIZONTAL 或 JSlider.VERTICAL。
- paintLabels** —— 如果为 true，则在间隔标记处绘制标签，但仅当设置了主要或次要间隔标记中间的间隙时才绘制标签。
- paintTicks** —— 如果为 true，则绘制间隔标记。但仅当设置了主要或次要间隔标记中间的间隙时才绘制间隔标记。
- snapToTicks** —— 如果为 true，则滑柄对应间隔标记。
- value** —— 由滑柄位置表示的当前值。
- ValuesIsAdjusting** —— 当拖动滑柄时为 true。

11.2.7 JSlider 事件

与大多数 Swing 组件一样，当修改 JSlider 的模型属性时，JSlider 实例激发变化事件，如表 11-2 所示。

图 11-13 中示出的小应用程序在它包含的滑杆上添加了一个变化监听器，以响应滑杆变化

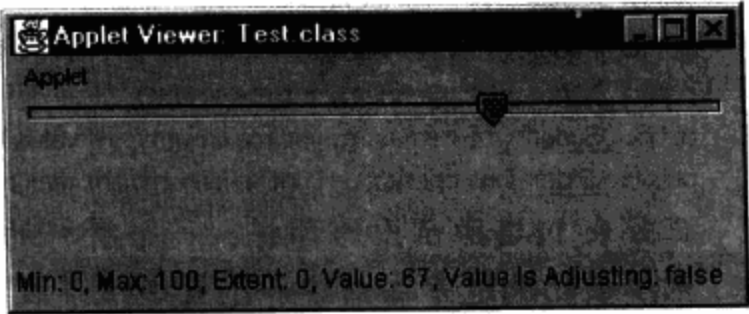


图 11-13 监视一个滑杆的值

事件。这个小应用程序还在状态区中显示滑杆的最小值、最大值和当前值。

例 11-11 中列出了图 11-13 中示出的小应用程序的代码。

例 11-11 监视一个滑杆的值

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JSlider slider = new JSlider ();

        contentPane.add (slider, BorderLayout.NORTH);

        slider.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                JSlider s = (JSlider) e.getSource ();
                showStatus ("Min: " + s.getMinimum () +
                    ", Max: " + s.getMaximum () +
                    ", Extent: " + s.getExtent () +
                    ", Value: " + s.getValue () +
                    ", Value Is Adjusting: " +
                    s.getValueIsAdjusting ());
            }
        });
    }
}
```

这个小应用程序还显示了滑杆的 `valueIsAdjusting` 属性，当拖动滑柄时，它的值为 `true`。在某些情况中，需要在滑柄不动时响应滑杆事件，而不是不断地对拖动滑柄进行响应。

11.2.8 JSlider 类总结

类总结 11-2 中列出了 `JSlider` 类的 `public` 和 `protected` 变量和方法。

类总结 11-2 JSlider

扩展： `JComponent`

实现： `SwingConstants`, `javax.accessibility.Accessible`

1. 构造方法

```
public JSlider ()
public JSlider (int orientation)
public JSlider (int minimum, int maximum)
public JSlider (int minimum, int maximum, int value)
public JSlider (int orientation, int minimum, int maximum, int value)
```

正如表 11-2 所示出的那样，一个滑杆的方向、值、最小值和最大值都可以在创建时刻指定。无参数的构造方法创建一个最小值、最大值和初始值分别为 0、100 和 50 的水平滑杆。

2. 方法

(1) 变化监听器/激发变化事件

```
public void addChangeListener (ChangeListener)
```

```
public void removeChangeListener (ChangeListener)
protected ChangeListener createChangeListener ()
protected void fireStateChanged ()
```

用上面列出的 `public addChangeListener` 和 `removeChangeListener` 方法向 JSlider 实例登记变化监听器。

与大多数 Swing 组件一样, JSlider 实现一个 private 监听器类——JSlider.ModelListener, 这个监听器监听滑杆模型激发的变化事件。CreateChangeListener 方法创建一个 JSlider.ModelListener 实例, 并向滑杆的模型登记。通过调用 fireStateChanged 方法把滑杆的模型激发的变化事件传递给滑杆的变化监听器。只有在极少情况中, 这种缺省行为才是不合适的, 这时可以在 JSlider 的扩展中重载这两个方法。

(2) 标准标签

```
public Hashtable createStandardLabels (int increment)
public Hashtable createStandardLabels (int increment, int start)
```

上面列出的两个方法为 JSlider 实例产生标准的标签。第一个方法带一个表示标签中间的递增的 integer 值。例如, 如果 createStandardLabels (int) 带入的值为 5, 且滑杆的最小值和最大值分别为 0 和 20, 那么, 将分别在值 0、5、10、15 和 20 处创建标签。

上面列出的第二个方法顺序地带入了一个递增值和一个开始值。例如为一个最小值和最小值分别为 0 和 20 的滑杆调用 createStandardLabels (5, 3) 将分别在 3、8、13 和 18 处产生标签。

(3) 属性访问方法

```
public int getExtent ()
public boolean getInverted ()
public Dictionary getLabelTable ()
public int getMajorTickSpacing ()
public int getMaximum ()
public int getMinimum ()
public int getMinorTickSpacing ()
public BoundedRangeModel getModel ()
public int getOrientation ()
public boolean getPaintLabels ()
public boolean getPaintTicks ()
public boolean getSnapToTicks ()
public int getValue ()
public boolean getValuesAdjusting ()

public void setExtent (int)
public void setInverted (boolean)
public void setLabelTable (Dictionary)
public void setMajorTickSpacing (int)
public void setMaximum (int)
public void setMinimum (int)
public void setMinorTickSpacing (int)
public void setModel (BoundedRangeModel)
public void setOrientation (int)
public void setPaintLabels (boolean)
public void setPaintTicks (boolean)
public void setSnapToTicks (boolean)
public void setValue (int)
public void setValuesAdjusting (boolean)
```

上面列出的方法是 JSlider 属性的访问方法。有关这些属性的更多信息请参见表 11-2。

(4) 方便的方法

```
public String toString ()
```

```
protected void updateLabelUIs ()
```

toString 方法返回一个字符串，这个字符串概括了 JSlider 属性及它们的值。当更改界面样式时，调用 updateLabelUIs 方法来更新滑杆的标签所使用的 JLabel 实例的 UI 代表。updateLabelUIs 方法极少被 JSlider 的扩展重载。

(5) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
```

```
public SliderUI getUI ()
```

```
public String getUIClassID ()
```

```
public void setUI (SliderUI)
```

```
public void updateUI ()
```

上面方法可以在 JComponent 的大多数扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

11.2.9 AWT 兼容

AWT 没有提供与 JSlider 类似的组件。

11.3 JSeparator

分隔条通常用于分隔（或组织）组件或组件组。尽管有人说分隔条是 Swing 的最简单组件[⊖]，但如果对布局管理器及组件的最小尺寸、最大尺寸和首选尺寸没有很好的了解，则在使用它时也有很多困惑。例如，可以考察一下图 11-14 中示出的小应用程序，这个小应用程序包含了两个由一个 JSeparator 实例分隔的按钮。

例 11-12 中列出了图 11-14 中示出的小应用程序的代码。

例 11-12 一个不可见的分隔条

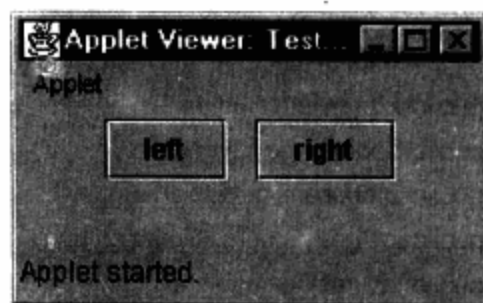


图 11-14 一个不可见的分隔条

```
import java.awt.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (new JButton ("left"));
        contentPane.add (new Jseparator (Jseparator.VERTICAL));
        contentPane.add (new JButton ("right"));
    }
}
```

这个小应用程序把一个 FlowLayout 实例设置为它的内容窗格的布局管理器，此后还把一个

⊖ JPanel 也引起了这种争论。

按钮、一个分隔条和另一个按钮添加到其内容窗格中。

很明显，这个小应用程序有点问题，因为在图 11-14 中，分隔条是不可见的。分隔条确实在这个小应用程序中，但它是不可见的，因为它的高度是 0；这就是原因。

缺省情况下，一个分隔条的 UI 代表把垂直分隔条的首选大小定义为 `new Dimension (2, 0)`。假定分隔条将由一个布局管理器来确定大小，或者，当它包含在一个带有一个 `null` 布局管理器的容器中时将显式地设置大小。由于这个小应用程序的内容窗格是一个 `FlowLayout` 实例，因此，这个分隔条的大小将由它的首选大小确定，所以它的高度设置为 0。

使这个分隔条可见的解决方法是简单地修改它的首选大小，正如图 11-15 中示出的小应用程序那样。

例 11-13 中列出了图 11-15 中示出的小应用程序的代码。

例 11-13 控制分隔条的大小

```
import java.awt.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane () ;
        JSeparator s = new JSeparator (JSeparator.VERTICAL);
        Dimension ps = s.getPreferredSize () ;
        contentPane.setLayout (new FlowLayout ()) ;
        contentPane.add (new JButton ("left"));
        contentPane.add (s);
        contentPane.add (new JButton ("right"));
        s.setPreferredSize (new Dimension (ps.width, 50));
    }
}
```

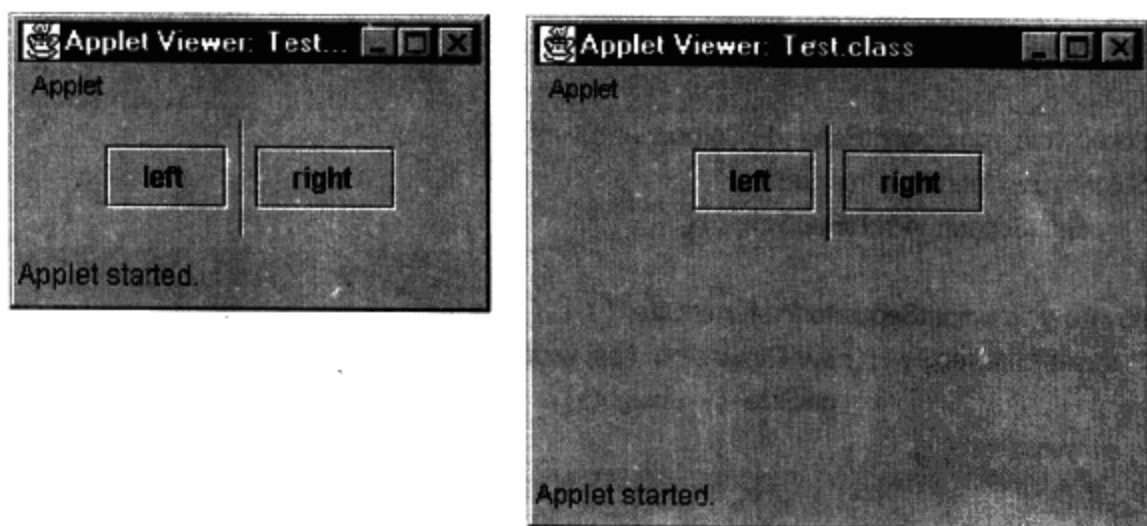


图 11-15 控制分隔条的大小

这个小应用程序不改变分隔条的首选宽度；但是把这个分隔条的首选高度设置为 50 个像素点。

注意观察图 11-15 中右图，这个分隔条的高度是固定的。常常需要分隔条的大小能够随它们所在的容器一起变化，如图 11-16 所示。

例 11-14 中列出了图 11-16 中示出的小应用程序的代码。

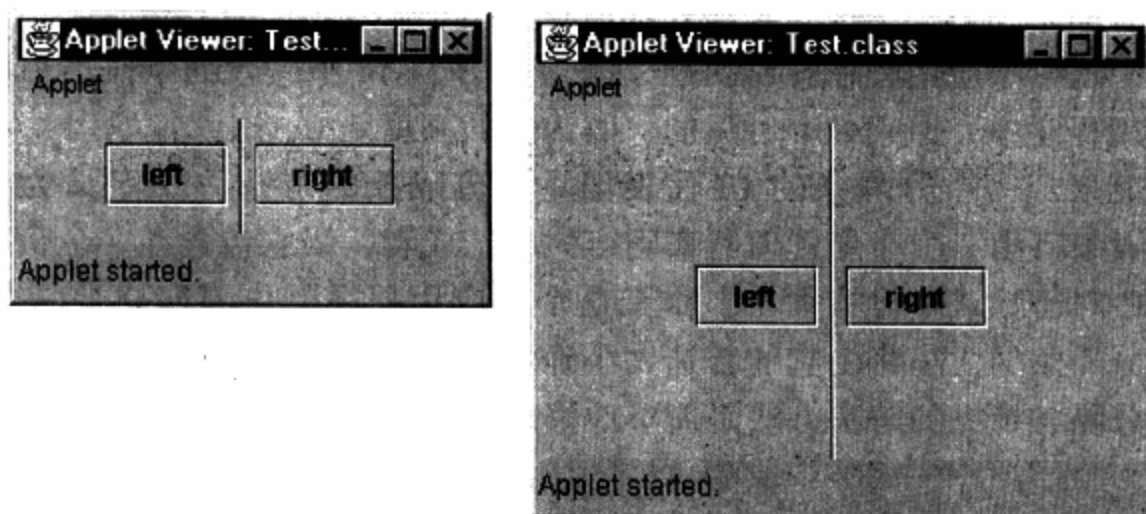


图 11-16 监控分隔条的大小

例 11-14 监控分隔条的大小

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        final JSeparator s =
            new JSeparator (JSeparator.VERTICAL);
        final Dimension ps = s.getPreferredSize ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (new JButton ("left"));
        contentPane.add (s);
        contentPane.add (new JButton ("right"));

        addComponentListener (new ComponentAdapter () {
            public void componentShown (ComponentEvent e) {
                adjustSeparatorPreferredSize ();
            }
            public void componentResized (ComponentEvent e) {
                System.out.println ("resized");
                adjustSeparatorPreferredSize ();
            }
            private void adjustSeparatorPreferredSize () {
                s.setPreferredSize (new Dimension (ps.width,
                    getSize ().height));
                s.revalidate ();
            }
        });
    }
}
```

图 11-16 中示出的小应用程序添加了一个组件监听器，这个监听器在内容窗格显示时或改变大小时调整分隔条的首选大小。在设置分隔条的首选大小后，为这个分隔条调用了 `revalidate()` 方法，以便强制性地把它绘制出来。有关 `revalidate` 方法的详细内容参见 4.3.5 节的“Validate、Invalidate 和 Revalidate”。

11.3.1 分隔条与框

分隔条非常适合于 Swing 的框容器——Box 类的实例，因为框可在水平方向上或在垂直方向上排列组件。一般来说，垂直分隔条放在水平框中，而水平分隔条放在垂直框中。6.5.2 节“Box 类”中讨论了 Box 类。

图 11-17 中示出的小应用程序说明了分隔条在嵌套框中的放置方式。这个小应用程序的内容窗格包含一个垂直方向的框（Left 面板），该框 10 个像素点宽，旁边有一个垂直分隔条。

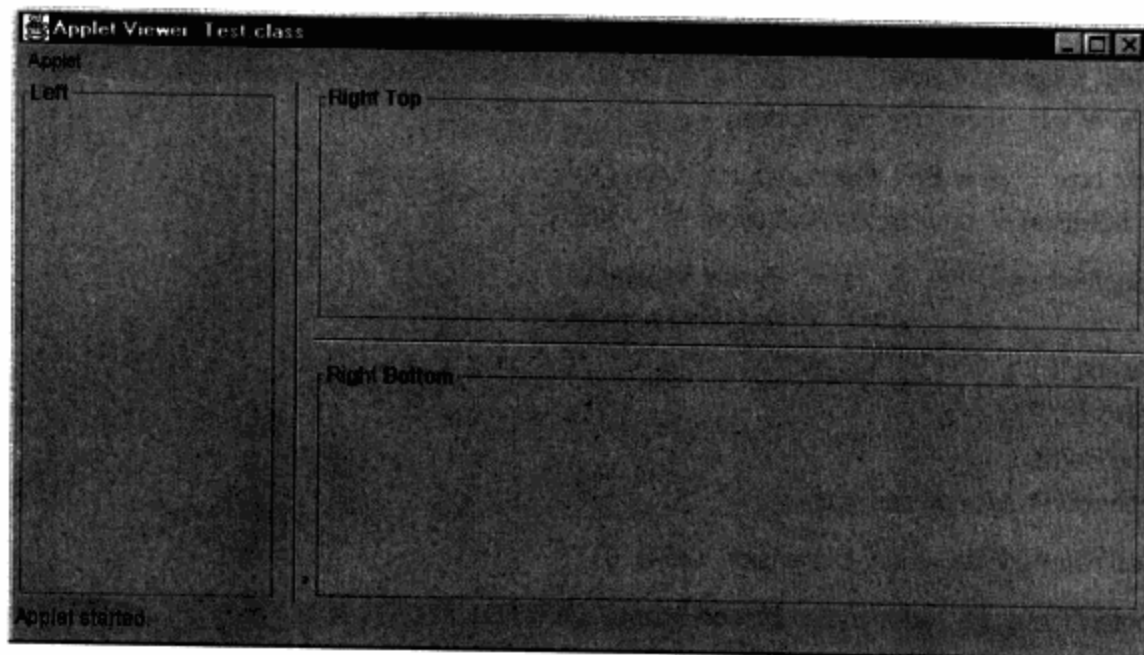


图 11-17 分隔条与框

第二个框是一个面板（Right Top 面板），该面板 10 个像素点高，下面有一个水平分隔条，下面是另一个面板（Right Bottom 面板）。

垂直分隔条的最大宽度被设置为这个分隔条的首选宽度，缺省是 2 个像素点。垂直分隔条的最大高度设置为 `Integer.MAX_VALUE`，以便这个分隔条能够在垂直方向上增大。

```
public class Test extends JApplet {
    ...
    private JSeparator vs, hs; // vs = vertical separator
                                // hs = horizontal separator
    ...
    public void init () {
        ...
        setSeparatorPreferredSize ();
        ...
    }
    private void setSeparatorPreferredSize () {
        vs. setMaximumSize (
            new Dimension (vs.getPreferredSize ().width,
                           Integer.MAX_VALUE));
        ...
    }
}
```

水平分隔条的最大高度设置为这个分隔条的首选高度，缺省是 2 个像素点。水平分隔条的最大宽度设置为 `Integer.MAX_VALUE`，以便这个分隔条能够在水平方向上增大。

```
...
hs. setMaximumSize (
    new Dimension (Integer.MAX_VALUE,
                   hs.getPreferredSize ().height));
```

例 11-15 中完整地列出了图 11-17 中示出的小应用程序的代码。

例 11-15 分隔条与框

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JPanel left, rightTop, rightBottom;
    private Box box = new Box (BoxLayout.X_AXIS),
        rightBox = new Box (BoxLayout.Y_AXIS);
    private JSeparator vs, hs; // vs = vertical separator
                                // hs = horizontal separator

    public void init () {
        createBoxes ();
        setPanelBorders ();
        setSeparatorPreferredSizes ();

        left.setPreferredSize (new Dimension (150, 0));
        getContentPane ().add (box, BorderLayout.CENTER);
    }

    private void createBoxes () {
        Component vStrut = box.createVerticalStrut (10),
            hStrut = box.createHorizontalStrut (10);

        rightBox.add (rightTop = new JPanel ());
        rightBox.add (box.createVerticalStrut (10));
        rightBox.add (hs = new JSeparator ());
        rightBox.add (box.createVerticalStrut (10));
        rightBox.add (rightBottom = new JPanel ());

        box.add (left = new JPanel ());
        box.add (box.createHorizontalStrut (10));
        box.add (vs = new JSeparator (JSeparator.VERTICAL));
        box.add (box.createHorizontalStrut (10));
        box.add (rightBox);
    }

    private void setSeparatorPreferredSizes () {
        vs.setMaximumSize (
            new Dimension (vs.getPreferredSize ().width,
                Integer.MAX_VALUE));

        hs.setMaximumSize (
            new Dimension (Integer.MAX_VALUE,
                hs.getPreferredSize ().height));
    }

    private void setPanelBorders () {
        left.setBorder (
            BorderFactory.createTitledBorder ("Left"));
        rightTop.setBorder (
            BorderFactory.createTitledBorder ("Right Top"));
        rightBottom.setBorder (
```

```
BorderFactory.createTitledBorder ("Right Bottom"));
```

组件总结 11-3 对 JSeparator 类进行了总结。

组件总结 11-3 JSeparator

模型：——
 UI 代表： javax.swing.plaf.basic.BasicSparatorUI
 绘制器：——
 编辑器：——
 激发的事件：——
 替代：——
 类图：

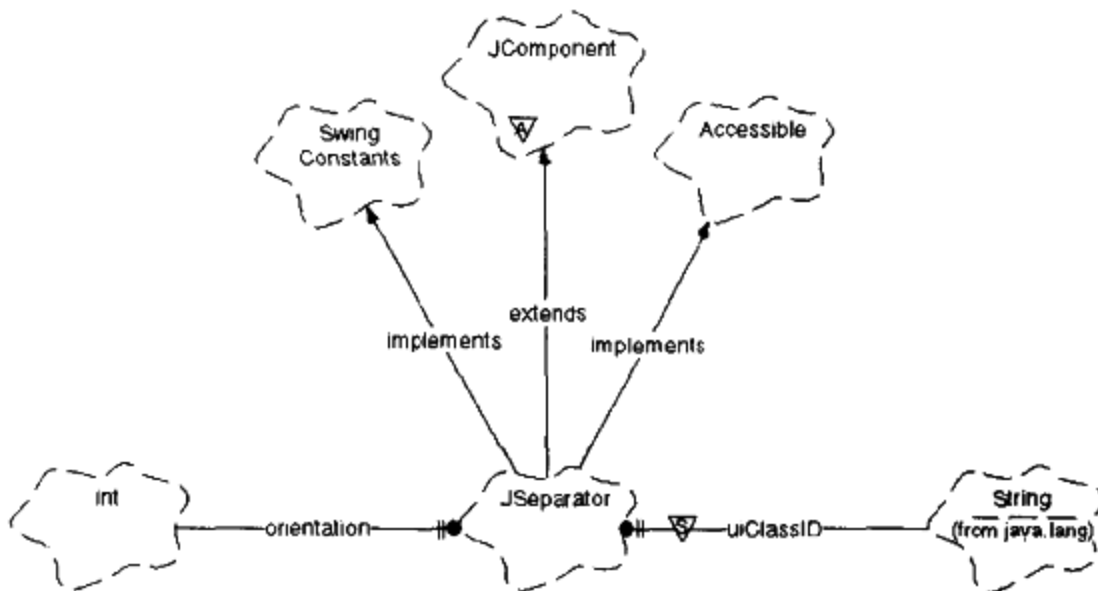


图 11-18 JSeparator 的类图

JSeparator 类扩展 JComponent，实现 SwingConstants 和 Accessible 接口。JSeparator 维护对一个代表分隔条的方向的 integer 值的 protected 引用和对一个字符串的引用，这个字符串代表分隔条的 UI 代表的类 ID。

11.3.2 JSeparator 属性

方向是 JSeparator 类维护的唯一属性。这个 orientation 属性是一个关联属性，JSeparator 为它提供了设置方法和获取方法[○]。当修改一个分隔条的方向时，将调用这个分隔条的 revalidate () 方法，重新布局并重绘这个分隔条。有关 revalidate () 方法的详细内容参见 4.3.5 节。

11.3.3 JSeparator 事件

JSeparator 没有它自己需要的事件；所有激发的事件都是从 JComponent 类继承的。

类总结 11-3 中对 JSeparator 类进行了总结。

类总结 11-3 JSeparator

[○] 方向也能在创建时刻指定。

扩展: JComponent

实现: SwingConstants, javax.accessibility.Accessible

1. 构造方法

```
public JSeparator ()  
public JSeparator (int orientation)
```

JSeparator 的实例可以用一个特定的方向值创建, 分隔条还可以用缺省的 JSeparator 构造方法创建。缺省的 JSeparator 构造方法创建一个水平分隔条。

2. 方法

(1) 方向/焦点可否决性/参数字符串

```
public int getOrientation ()  
public void setOrientation (int orientation)  
  
public boolean isFocusTraversable ()  
public String paramString ()
```

JSeparator 为 orientation 属性提供了设置方法和获取方法。另外, JSeparator 重载 isFocusTraversable(), 以返回 false, 表示分隔条不必获得焦点。paramString() 方法返回一个表示分隔条的字符串。

(2) 可访问性/插入式界面样式

```
public AccessibleContext getAccessible ()  
public SeparatorUI getUI ()  
public String getUIClassID ()  
public void setUI (SeparatorUI)  
public void updateUI ()
```

上面方法可以在 JComponent 的大多数扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

11.3.4 AWT 兼容

AWT 没有提供与 JSeparator 类似的组件。

11.4 本章回顾

进度条、滑杆和分隔条明显地是 AWT 的遗漏, 在 Swing 组件集中增加进度条、滑杆和分隔条是大受欢迎的。尽管进度条和滑杆可以认为是特殊的组件, 不属于用户接口工具包中的基本部分, 但分隔条却是最基本的用户接口工具之一。Swing 提供的这三个组件功能丰富、强大, 必将在很多 Java 小应用程序和应用程序中发挥作用。

第 12 章 轻量容器

本章讨论如下 Swing 轻量容器：

- JPanel
- JRootPane
- JLayeredPane
- JTabbedPane
- JSplitPane

JPanel 是 AWT 的 Canvas 和 Panel 类的继承类。文本和图形都可以绘制到 JPanel 实例中，并且 JPanel 实例可以用作一个通用容器。

JRootPane 是一个包含在窗体、对话框、窗口、内部窗体和 Swing 小应用程序等 Swing 顶层容器中的容器。

JLayeredPane 允许把它所包含的组件放置在不同的层中。层控制显示组件的深度。

JTabbedPane 是一个能够包含多个组件的容器。JTabbedPane 包含的多个组件一次只能显示一个。JTabbedPane 的实例包含能够用于选取当前显示的组件的选项卡。

JSplitPane 包含两个组件，这两个组件由一个分隔体所分隔。可以拖动分隔体以改变每个组件所占据的空间大小。

12.1 JPanel

JPanel 是最简单的 Swing 组件之一；但它也是使用最多的组件之一。Swing 在很多其他组件中使用了 JPanel 实例；例如，缺省时，JRootPane 容器为它的内容窗格和玻璃窗格创建了 JPanel 实例，正如表 12-1 所反映的那样。

JPanel 类具有简单容器和显示图形的画布的双重功能。图 12-1 示出的小应用程序创建了一个 JPanel 实例：一个包含“Name:”选项卡和文本域的控制面板、一个专门用作显示文本和图形的画布的窗格，以及包含上述控制面板和画布的第三个面板。

例 12-1 中列出了图 12-1 中示出的小应用程序的代码。

例 12-1 一个使用了三个 JPanel 实例的小应用程序

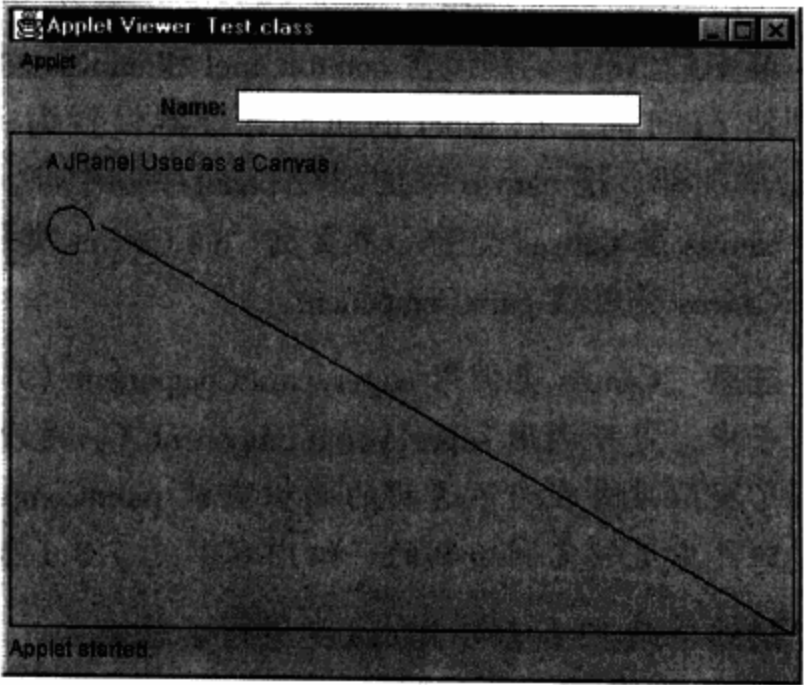


图 12-1 用作容器和画布的面板

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;
```



```

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JPanel panel = new JPanel (new BorderLayout ());
        JPanel controlPanel = new JPanel ();
        JPanel canvas = new Canvas ();

        canvas.setBorder (
            BorderFactory.createLineBorder (Color.black));

        controlPanel.add (new JLabel ("Name:"));
        controlPanel.add (new JTextField (20));

        panel.add (controlPanel, BorderLayout.NORTH);
        panel.add (canvas, BorderLayout.CENTER);

        contentPane.add (panel);
    }
}

class Canvas extends JPanel {
    public void paintComponent (Graphics g) {
        super.paintComponent (g);

        Dimension size = getSize ();
        g.setColor (Color.black);
        g.drawLine (50, 50, size.width, size.height);
        g.drawArc (20, 40, 25, 25, 0, 290);
        g.drawString ("A JPanel Used as a Canvas", 20, 20);
    }
}

```

其中的 panel 实例包含 controlPanel 和 canvas 面板，并添加到这个小应用程序的内容窗格中。在创建 panel 时，把 panel 的布局管理器设置为一个 BorderLayout 实例；把 controlPanel 添加到 panel 中上部，把 canvas 添加到在 panel 中的中部。

canvas 是 Canvas 类的一个实例，而 Canvas 类是 JPanel 的一个扩展。为了绘制它的文本和图形，Canvas 类重载 paintComponent。

注意 Canvas 类调用 super.paintComponent () 来确保绘制它的背景。在这个简单的例子中，是否调用 super.paintComponent () 是没有区别的，但例子中包含这个调用是为了提醒读者在为不透明的面板重载 paintComponent 时要调用超类的方法。有关在 Swing 组件中进行定制绘制的详细内容参见 4.3.1 节“在 Swing 组件中的定制绘制”。

组件总结 12-1 中对 JPanel 类进行了总结。

组件总结 12-1 JPanel

模型:	——
UI 代表:	javax.swing.plaf.basic.BasicPanelUI
绘制器:	——
编辑器:	——
激发的事件:	——
替代:	java.awt.Canvas、java.awt.Panel
类图:	见图 12-2

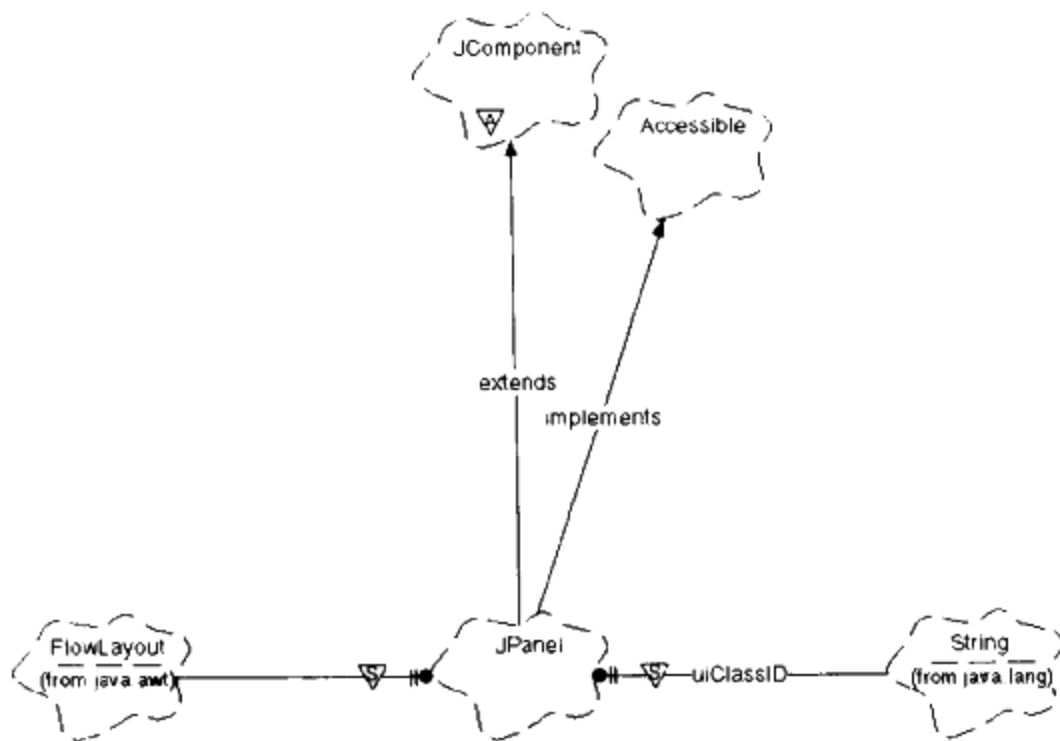


图 12-2 JPanel 的类图

JPanel 是 JComponent 的简单扩展，它没有模型。JPanel 仅比它的缺省布局管理器多做了一点事情，双缓存和不透明属性，以及实现 Accessible 接口。缺省情况下，把 JPanel 实例的布局管理器设置为 FlowLayout 的一个共享实例，并且继承的双缓存和不透明属性都设置为 true。

注意 JPanel 本身并不为它所包含的组件实现任何功能。由于 JComponent 扩展 java.awt.Container 类，所以所有 Swing 轻量组件都是具有完全功能的 AWT 容器。

Swing 提示

JPanel 的缺省值

JPanel 是最简单的 Swing 组件之一，因此它非常易于使用。对 JPanel 实例要注意的重要事情是布局管理器、双缓存和不透明属性的缺省值。下面列出了这些缺省值：

布局管理器：FlowLayout 的一个共享实例；双缓存：True；不透明：True。

12.1.1 JPanel 的属性

JPanel 没有定义任何它自己的属性；所有的属性都是从 JComponent 类继承的。

12.1.2 JPanel 的事件

JPanel 没有它自己需要的事件；所有激发的事件都是从 JComponent 类继承的。

12.1.3 JPanel 类总结

类总结 12-1 中列出了 JPanel 类的 public 和 protected 变量和方法。

类总结 12-1 JPanel

扩展：JComponent

实现：javax.accessibility.Accessible

1. 构造方法

```
public JPanel ()
public JPanel (boolean isDoubleBuffered)
public JPanel (LayoutManager)
public JPanel (LayoutManager, boolean isDoubleBuffered)
```

JPanel 提供了上面列出的四个构造方法。其中的无参数构造方法为面板配备了 FlowLayout 的一个共享实例，并把双缓存和不透明属性设置为 true。另外三个构造方法可用于为面板指定双缓存属性 (boolean 值) 和布局管理器。

2. 方法

```
public AccessibleContext getAccessibleContext ()
public String getUIClassID ()
public String paramString ()
public void updateUI ()
```

JPanel 的方法是对继承得到的方法的简单重载，这些继承方法用来确认 JPanel 实例。

12.1.4 AWT 兼容

JPanel 与 java.awt.Panel 是完全源代码兼容的，但这不是绝对的，因为 java.awt.Panel 没有定义任何它自己的 public 方法[○]。JPanel，与 JComponent 一样，提供了双缓存和不透明属性，在 AWT 的 Panel 类中没有它们的对等属性。

12.2 JRootPane

几乎每一个 Swing 组件都最终处在一个 JRootPane 实例中，因为所有 Swing 顶层容器都包含一个 JRootPane 实例。根容器提供了一种方便的包容结构。

一个根窗格中的最顶层组件是玻璃窗格。当玻璃窗格可见时，它浮到根窗格中的所有组件之上并截获所有的鼠标事件。

在玻璃窗格之下是一个 JLayeredPane 实例，正如其名称所提示的那样，该实例把组件放到不同的层上。分层窗格又进一步包含一个可选的菜单栏和一个内容窗格。内容窗格是小应用程序和应用程序的应用基础。

Swing 为包含一个 JRootPane 实例的容器定义了一个接口。RootPaneContainer 接口将在下面的“RootPaneContainer 接口”小节中讨论。

表 12-1 列出了 JRootPane 实例中所包含的组件

表 12-1 JRootPane 包含的组件

组件	类	描述
contentPane	JPanel	包含小应用程序/应用程序的组件，驻留在分层窗格中
glassPane	JPanel	浮在所有其他组件之上，截获鼠标事件并且可以是透明的
layeredPane	JLayeredPane	包含内容窗格和菜单栏
menuBar	JMenuBar	驻留在内容窗格之上的分层窗格中

12.2.1 RootPaneContainer 接口

接口总结 12-1 中对 RootPaneContainer 接口进行了总结。

○ 被重载的 addNotify 方法除外。

接口总结 12-1 RootPaneContainer

```

public abstract Container getContentPane ()
public abstract Component getGlassPane ()
public abstract JLayeredPane getLayeredPane ()
public abstract JRootPane getRootPane ()
public abstract void setContentPane (Container)
public abstract void setGlassPane (Component)
public abstract void setLayeredPane (JLayeredPane)

```

RootPaneContainer 接口为根窗格本身及包含在根窗格中玻璃窗格、分层窗格和内容窗格定义了访问方法。RootPaneContainer 接口是用下述 Swing 容器定义的：

- JApplet
- JDialog
- JFrame
- JInternalFrame
- JRootPane
- JWindow

图 12-3 示出了一个类图，其中描述了 RootPaneContainer 接口与 Swing 容器之间的关系。

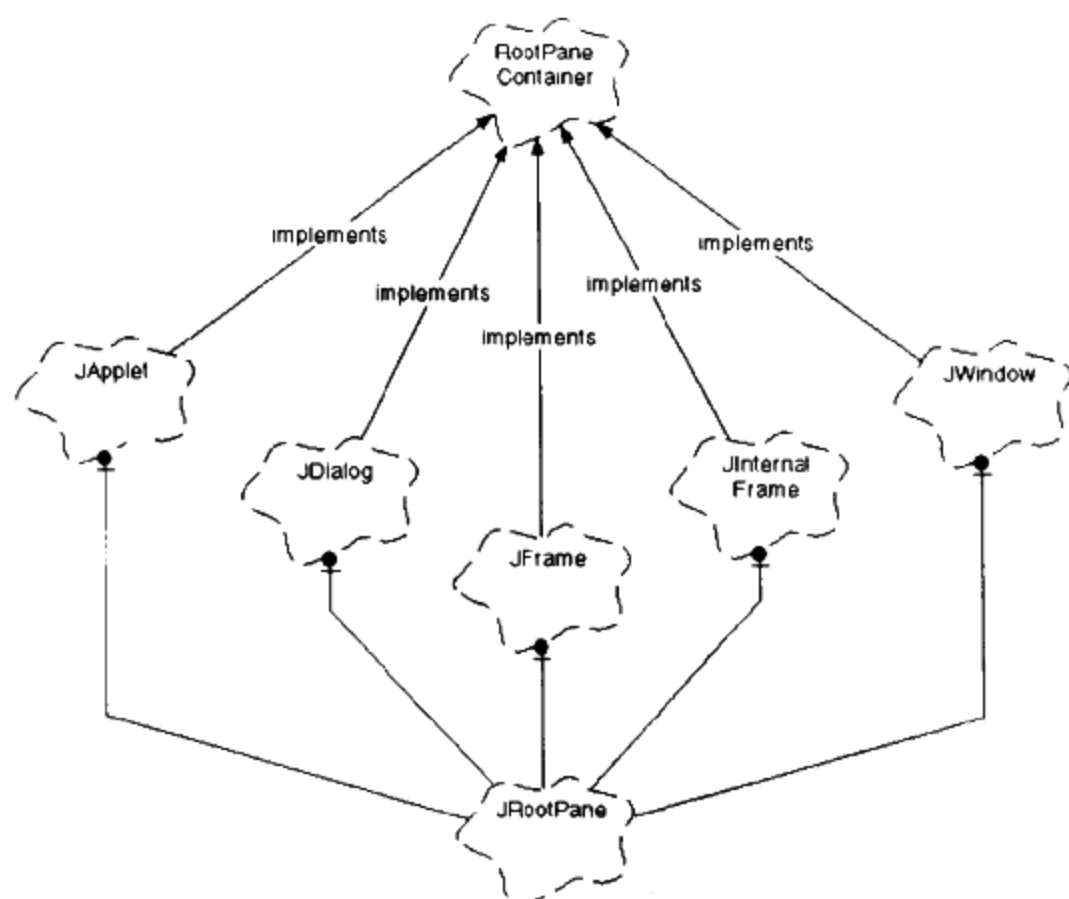


图 12-3 RootPaneContainer 接口

每个实现 RootPaneContainer 接口的 Swing 组件都维护一个对 JRootPane 实例的引用。由 RootPaneContainer 接口定义的方法是通过委托给相应的根窗格来实现的。

12.2.2 玻璃窗格

玻璃窗格在所有根窗格组件中是最有趣的，因为玻璃窗格浮在其他组件之上，截获根窗格中的鼠标事件，并且可以是透明的。

这些独特的功能使玻璃窗格具有广泛的特殊用途。从拖放式动画到蟑螂“工具”（小蟑螂

在打开和关闭窗口时快速跑动)都适用于揭示玻璃窗格的用途。

玻璃窗格浮在一个根窗格中的所有其他组件之上,因为玻璃窗格是添加到一个根窗格中的第一个组件。

Swing 组件的层序或深度是由它们添加到各自容器中的顺序决定的。在同一个容器中,第一个加入到这个容器中的组件在所有其他组件之上显示,而最后添加到这个容器中的组件显示在所有其他组件之下。有关轻量组件和重量组件的层序的确定方式的详细内容参见 2.3.1 节“层序”。

当玻璃窗格是可见时,它们还能够截获鼠标事件。当一个鼠标事件在一个 AWT 容器中发生时,这个事件传递给希望获取鼠标事件的最上层轻量组件。如果一个组件安装了一个鼠标监听器或特别许可了鼠标事件,那么它就是希望获取鼠标事件的。

图 12-4 中示出的小应用程序包含了两个按钮:一个 JButton 实例和一个 JRadioButton 实例。其中的单选钮配备了一个光标进入的图像,这样,当鼠标进入这个按钮时就显示这个光标进入图像。有关按钮和光标进入图像的详细内容参见 8.4 节的“JButton”。当激活这个按钮时,与这个小应用程序的根窗格相关的玻璃窗格就变成可见的。此后,当玻璃窗格可见时,在这个小应用程序中的任何位置单击鼠标按钮都将使玻璃窗格隐藏起来,这个小应用程序也将复位成启动它时所处的状态。

图 12-4 示出了这个小应用程序的三种状态。上面的两幅图示出了这个小应用程序在玻璃窗格不可见时的样子;右边的图示出了鼠标进入单选钮后,这个按钮为光标进入时的图像。下图示出了“show glass pane”按钮被激活,且玻璃窗格可见后,这个小应用程序的样子。

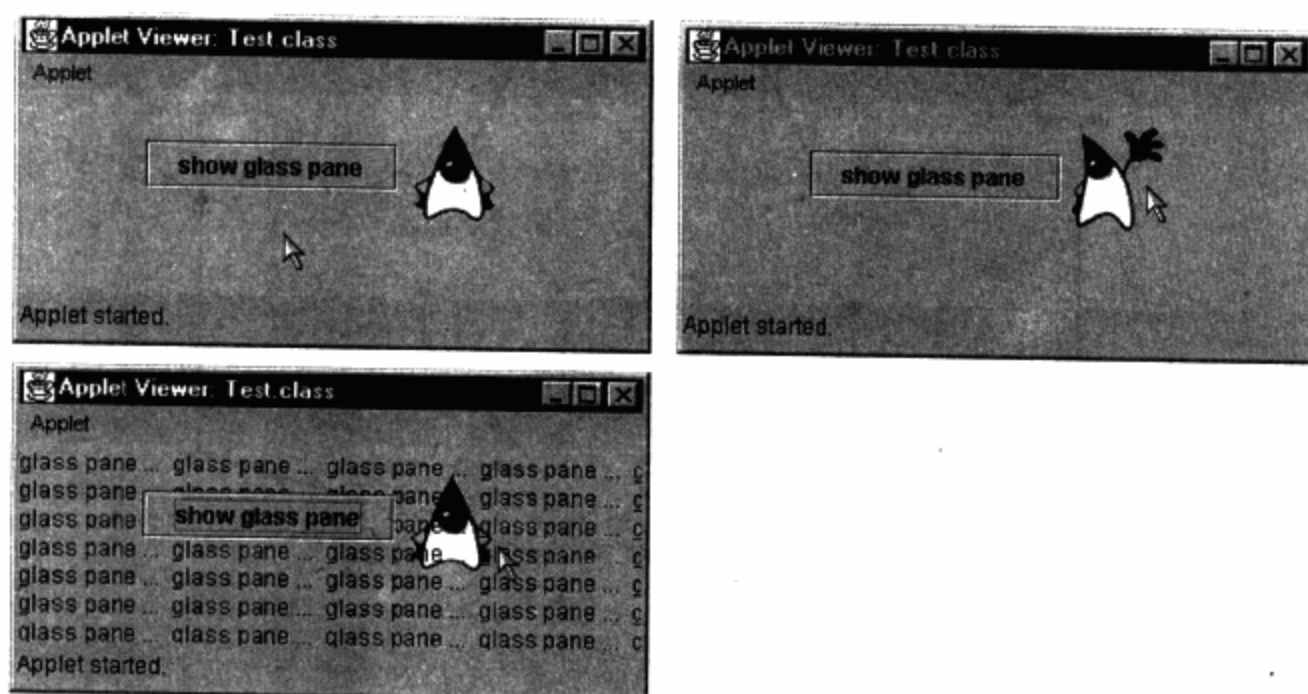


图 12-4 玻璃窗格

注意 当玻璃窗格可见时,单选钮的光标进入图像是不显示的,即便光标进入单选钮也是如此。这是因为玻璃窗格带有一个鼠标监听器,这表示它是希望处理鼠标事件的。由于 AWT 的代表事件模型不自动把事件从容器传递给它们包含的组件,因此这个小应用程序把鼠标事件传递给了玻璃窗格,但不传递给玻璃窗格下面的组件。

这个小应用程序用一个定制版本(一个 CustomGlassPane 实例)替换了其最初的玻璃窗格。

```
public class GlassPaneTest extends JApplet {
    private Component glassPane = new CustomGlassPane ();
```

```

public void init () {
    ...
    setGlassPane (glassPane);
    ...
}

```

CustomGlassPane 扩展 JPanel，并重载 PaintComponent 方法。当这个玻璃窗格可见时，字符串“glass pane ...”平铺在其背景上。注意，CustomGlassPane. PaintComponent 方法不动用 supper. PaintComponent 方法，因为它是透明的。

```

class CustomGlassPane extends JPanel {
    ...
    private String displayString = "glass pane ...";
    public void paintComponent (Graphics g) {
        Dimension size = getSize ();
        FontMetrics fm = g.getFontMetrics ();
        int sw = fm.stringWidth (displayString);
        int fh = fm.getHeight ();
        g.setColor (Color.blue);
        for (int row = fh; row < size.height; row += fh)
            for (int col = 0; col < size.width; col += sw)
                g.drawString (displayString, col, row);
    }
    ...
}

```

CustomGlassPane 通过使其自身可见并允许内容窗格中的按钮处理事件来响应鼠标按下事件。

```

class CustomGlassPane extends JPanel {
    private JButton button;
    private String displayString = "glass pane ...";
    public CustomGlassPane () {
        setOpaque (false);
        addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                setVisible (false);
            }
        });
    }
    ...
}

```

例 12-2 中完整地列出了图 12-4 中示出的小应用程序的代码。

例 12-2 玻璃窗格测试小应用程序

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GlassPaneTest extends JApplet {
    private Component glassPane = new CustomGlassPane ();

```

```

public void init () {
    Container contentPane = getContentPane ();
    JButton button = new JButton ("show glass pane");
    contentPane.setLayout (new FlowLayout ());
    contentPane.add (button);
    setGlassPane (glassPane);
    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            glassPane.setVisible (true);
        }
    });
}

class CustomGlassPane extends JPanel {
    private JButton button;
    private String displayString = "glass pane ... ";

    public CustomGlassPane () {
        setOpaque (false);
        addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                setVisible (false);
            }
        });
    }

    public void paintComponent (Graphics g) {
        Dimension size = getSize ();
        FontMetrics fm = g.getFontMetrics ();
        int sw = fm.stringWidth (displayString);
        int fh = fm.getHeight ();
        g.setColor (Color.blue);
        for (int row = fh; row < size.height; row += fh)
            for (int col = 0; col < size.width; col += sw)
                g.drawString (displayString, col, row);
    }
}

```

Swing 提示

玻璃窗格与鼠标事件

当一个轻量容器中发生了一个鼠标事件时, 把这个事件传递给该容器中希望处理鼠标事件最顶层组件。因此, 如果根窗格的玻璃窗格可见且这个窗格表示希望处理鼠标事件, 则将鼠标事件传递给该窗格。组件可以用两种方法来表示希望处理鼠标事件, 即添加一个鼠标监听器或允许鼠标事件。允许鼠标事件通过以相应的 `AWTEvent` 常数调用允许事件方法, 如下所示:

```
aComponent.enableEvents (AWTEvent.MOUSE_EVENT_MASK);
```

如果一个玻璃窗格没有表示希望处理鼠标事件, 则把鼠标事件将传送给玻璃窗格之下的组件, 即使玻璃窗格是可见的。

12.2.3 内容窗格

内容窗格是小应用程序和应用程序放置它们的组件的地方。缺省情况下，内容窗格是一个用 `JRootPane.createContentPane` 方法配备了边框布局管理器的 `JPanel` 实例。

图 12-5 中示出的小应用程序实现了一个定制的内容窗格，它在背景上平铺一幅图像。

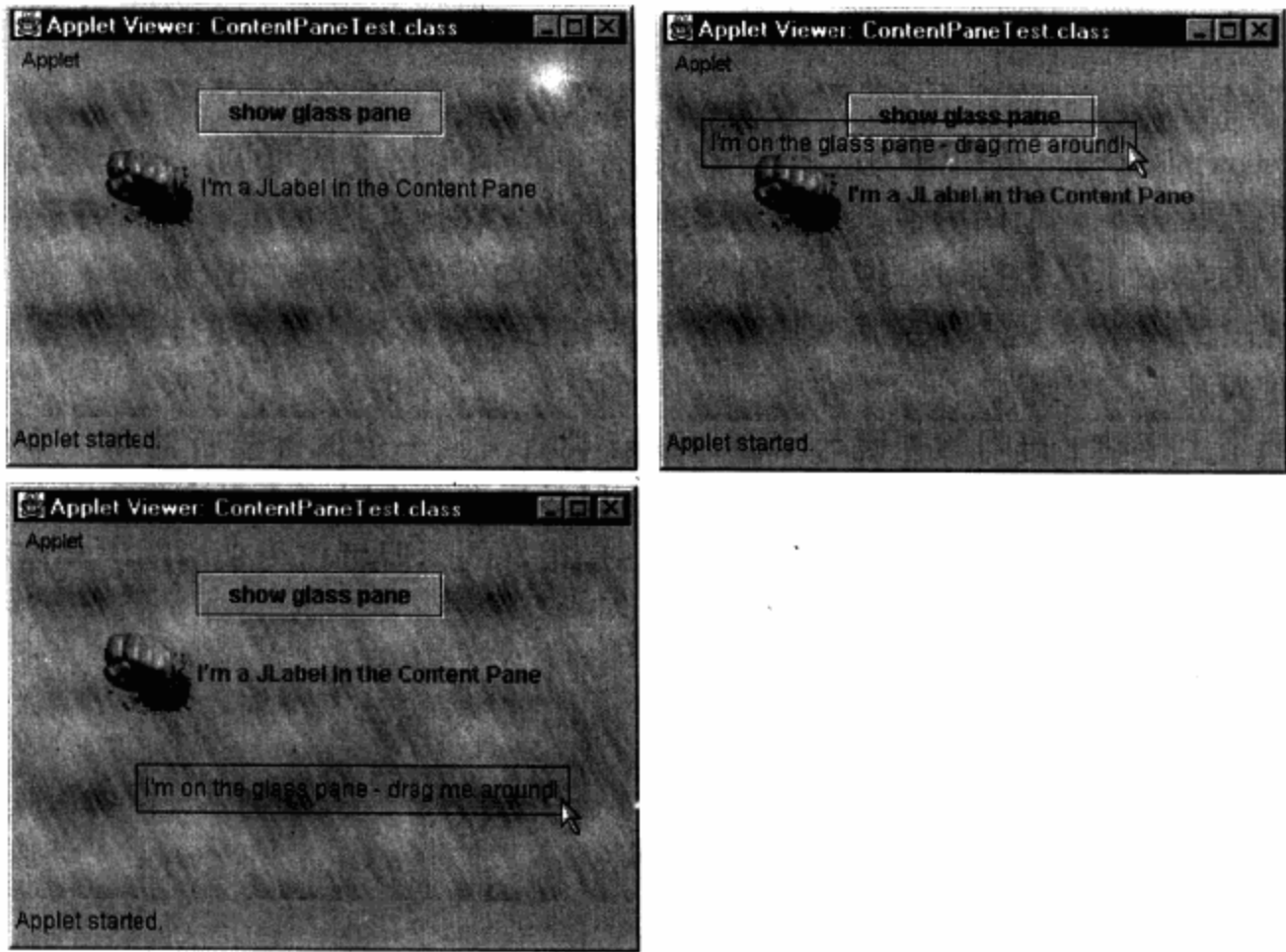


图 12-5 内容窗格

这个小应用程序用定制版本替换了它的玻璃窗格和内容窗格。

```
public class ContentPaneTest extends JApplet {
    private JButton button = new JButton ("show glass pane");

    public void init () {
        setGlassPane (new CustomGlassPane (button));
        setContentPane (new CustomContentPane (button));
        ...
    }
    ...
}
```

图 12-5 中左上图示出了这个小应用程序处于开始状态时的样子。这个小应用程序的内容窗格在它的背景上平铺 `rain.gif` 图像，还添加了一个选项卡和一个按钮。

```
class CustomContentPane extends JPanel {
    private ImageIcon rain = new ImageIcon ("rain.gif");
    private ImageIcon punch = new ImageIcon ("punch.gif");
    private int rainw = rain.getIconWidth ();
    private int rainh = rain.getIconHeight ();

    public CustomContentPane (JButton button) {
        add (button);
        add (new JLabel ("I'm a JLabel in the Content Pane",
```

```

        punch, SwingConstants.RIGHT));
    }

    public void paintComponent (Graphics g) {
        Dimension size = getSize ();

        // tile rain image over backgroundTo
        for (int row=0; row < size.height; row += rainh)
            for (int col=0; col < size.width; col += rainw)
                rain.paintIcon (this, g, col, row);
    }
}

```

当激活 show glass pane 按钮时，这个小应用程序的玻璃窗格变成可见。这个玻璃窗格包含一个可拖动的对象——一个四周带一个黑矩形的字符串。在拖动这个字符串后，该字符串就消失了。如果随后激活了这个按钮，这个字符串将在它拖动的最后位置上出现。

图 12-5 右上图示出了这个小应用程序在激活按钮且玻璃窗格可见后的样子。图 12-5 下图示出了拖动字符串后的样子。

在玻璃窗格上拖动字符串的关键与 CustomGlass 类有关。一个鼠标监听器和一个鼠标运动监听器控制字符串的拖动，在玻璃窗格中每次发生鼠标拖动事件时都更新字符串的位置并重新绘制它。应该指出的是，这种拖动字符串的方式不是很好的，但可以用来说明内容窗格的使用方法。

```

class CustomGlassPane extends JPanel {
    ...
    private Point ulhc = new Point (20, 20), last;
    private String displayString =
        "I'm on the glass pane - drag me around!";

    public CustomGlassPane (JButton b) {
        ...
        addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                last = e.getPoint ();
            }
            public void mouseReleased (MouseEvent e) {
                setVisible (false);
            }
        });
        addMouseMotionListener (new MouseMotionAdapter () {
            public void mouseDragged (MouseEvent e) {
                Point drag = e.getPoint ();
                ulhc.x += drag.x - last.x;
                ulhc.y += drag.y - last.y;

                repaint ();

                last.x = drag.x;
                last.y = drag.y;
            }
        });
        ...
    }
}

```

例 12-3 中完整地列出了图 12-5 中示出的小应用程序的代码。

例 12-3 一个定制的内容窗格

```

import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class ContentPaneTest extends JApplet {
    private JButton button = new JButton ("show glass pane");

    public void init () {
        setGlassPane (new CustomGlassPane (button));
        setContentPane (new CustomContentPane (button));

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                getGlassPane ().setVisible (true);
            }
        });
    }

    class CustomContentPane extends JPanel {
        private ImageIcon rain = new ImageIcon ("rain.gif");
        private ImageIcon punch = new ImageIcon ("punch.gif");
        private int rainw = rain.getIconWidth ();
        private int rainh = rain.getIconHeight ();

        public CustomContentPane (JButton button) {
            add (button);
            add (new JLabel ("I' m a JLabel in the Content Pane",
                punch, SwingConstants.RIGHT));
        }

        public void paintComponent (Graphics g) {
            Dimension size = getSize ();

            for (int row=0; row < size.height; row += rainh)
                for (int col=0; col < size.width; col += rainw)
                    rain.paintIcon (this, g, col, row);
        }
    }

    class CustomGlassPane extends JPanel {
        private JButton button;
        private Point ulhc = new Point (20, 20), last;
        private String displayString =
            "I' m on the glass pane - drag me around!";

        public CustomGlassPane (JButton b) {
            button = b;

            setOpaque (false);

            addMouseListener (new MouseAdapter () {
                public void mousePressed (MouseEvent e) {
                    last = e.getPoint ();
                }

                public void mouseReleased (MouseEvent e) {
                    setVisible (false);
                }
            });
        }
    }
}

```

```

addMouseListener (new MouseMotionAdapter () {
    public void mouseDragged (MouseEvent e) {
        Point drag = e.getPoint ();
        ulhc.x += drag.x - last.x;
        ulhc.y += drag.y - last.y;

        repaint ();

        last.x = drag.x;
        last.y = drag.y;
    }
});
}

public void paintComponent (Graphics g) {
    FontMetrics fm = g.getFontMetrics ();
    int sw = fm.stringWidth (displayString);
    int sh = fm.getHeight ();
    int ascent = fm.getAscent ();

    g.drawRect (ulhc.x, ulhc.y, sw + 10, sh + 10);
    g.drawString (displayString,
        ulhc.x + 5, ulhc.y + ascent + 5);
}
}

```

Swing 提示

内容窗格和玻璃窗格的布局管理器

缺省情况下，内容窗格和玻璃窗格都是面板 (JPanel 的实例)，它们是用 JRootPane 类中 protected createContentPane 和 createGlassPane 方法创建的。

用 JRootPane.createGlassPane 创建的玻璃窗格保留了 JPanel 实例的布局管理器——一个 FlowLayout 实例。另一方面，用 JRootPane.createContentPane 创建的内容窗格配备了一个 BorderLayout 实例。

组件总结 12-2 中对 JRootPane 类进行了总结。

组件总结 12-2 JRootPane

模型: _____
 UI 代表: _____
 绘制器: _____
 编辑器: _____
 激发的事件: _____
 替代: _____
 类图: 见图 12-6

JRootPane 是 JComponent 的一个简单扩展，并实现 Accessible 接口。与 JPanel 一样，JRootPane 没有模型或 UI 代表。JRootPane 维护对 JLayeredPane 和 JMenuBar 实例的引用。JRootPane 还维护一个对一个 JButton 实例的 protected 引用，这个 JButton 实例是根窗格的缺省按钮。



图 12-6 JRootPane 的类图

12.2 .4 JRootPane 属性

表 12-2 中列出了 JRootPane 维护的属性。

表 12-2 JRootPane 的属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
contentPane	Container	S	SG	JPanel ^④
defaultButton	JButton	B	SG	null
glassPane	Component	S	SG	JPanel
jMenuBar	JMenuBar	S	G	true
layeredPane	JLayeredPane	S	SG	null
validateRoot	JRootPane	S	G	JPanel

- ① B = 关联的 (激发 PropertyChangeEvent) / C = 受约束的 / I = 索引的 /
S = 简单的 / Ch = 激发 ChangeEvent
- ② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
- ③ I&F = 与界面样式有关
- ④ 缺省时，内容窗格的布局管理器是一个 BorderLayout
- contentPane** —— 一个小应用程序和应用程序在其中放置它们的组件的容器。
- defaultButton** —— 一个按钮。当根窗格中的任意组件具有焦点时，都可以用一个键击激活这个按钮。
- glassPane** —— 一个组件，它浮在一个根窗格中包含的所有组件之上。
- JMenuBar** —— 一个放在内容窗格中的菜单栏。
- layeredPane** —— 一个包含内容窗格和菜单栏的 JLayeredPane 实例。
- validateRoot** —— 如果一个 Swing 容器从 isValidateRoot 返回 true，那么当对该容器所包含的组件之一调用 revalidate () 时，这个容器将验证它的所有组件。JRootPane 获得 JScrollPane 是唯

一从 `isValidRoot()` 方法返回 `true` 的 Swing 容器。实际上,这意味着,对一个根窗格或滚动窗格包含的任意组件调用 `revalidate()` 方法将导致对根窗格或滚动窗格调用 `validate()` 方法。其效果是,不论何时对根窗格中的组件之一调用 `revalidate()` 方法将导致重新绘制根窗格或滚动窗格中的所有组件。

12.2.5 JRootPane 事件

除了响应对关联属性的修改而激发属性变化事件外, `JRootPane` 不激发自己的事件。

由于 `JRootPane` 是一种简单的容器,因此,从 `JRootPane` 包含的组件的角度讨论事件处理比讨论根窗格本身激发的事件要有意得多。

如前所述,包含在一个根窗格中的玻璃窗格将占用鼠标事件,如果这个玻璃窗格表示它希望处理鼠标事件的话。图 12-4 中示出并在例 12-2 讨论的小应用程序说明了这样一个事实,即玻璃窗格处理的鼠标事件不自动传递给玻璃窗格之下的组件。但是,有些时候,使玻璃窗格把事件传递给其下的组件会是带来方便的。

图 12-7 示出的小应用程序用玻璃窗格来显示对一个文档的注释。在本例中,这个文档是这个应用程序的源代码。这个小应用程序包含了一个复选框来反复切换玻璃窗格的可见性,因此,也就是注释的可见性。如果玻璃窗格占用鼠标事件,那么在第一次显示注释之后就没有办法取消注释了,因为在切换复选框上单击鼠标的事件将被玻璃窗格截获。由于这个原因,这个玻璃窗格则不表示希望处理鼠标事件。

这个小应用程序创建了一个 `AnnotationPane` 实例,它被用作根窗格的玻璃窗格。这个小应用程序还创建了一个用于切换玻璃窗格可见性的 `JCheckBox` 实例。图 12-7 中上图示出了这个小应用程序最开始时的样子,下图则示出了这个小应用程序在把玻璃窗格设置为可见的之后的样子。

在这个小应用程序的 `init` 方法中调用的两个 `private` 方法创建了这个小应用程序的容器的体系结构并创建了玻璃窗格。

```
public class Test extends JApplet {
    Component glassPane = new AnnotationPane ();
    JCheckBox annotations = new JCheckBox ("annotations");
```

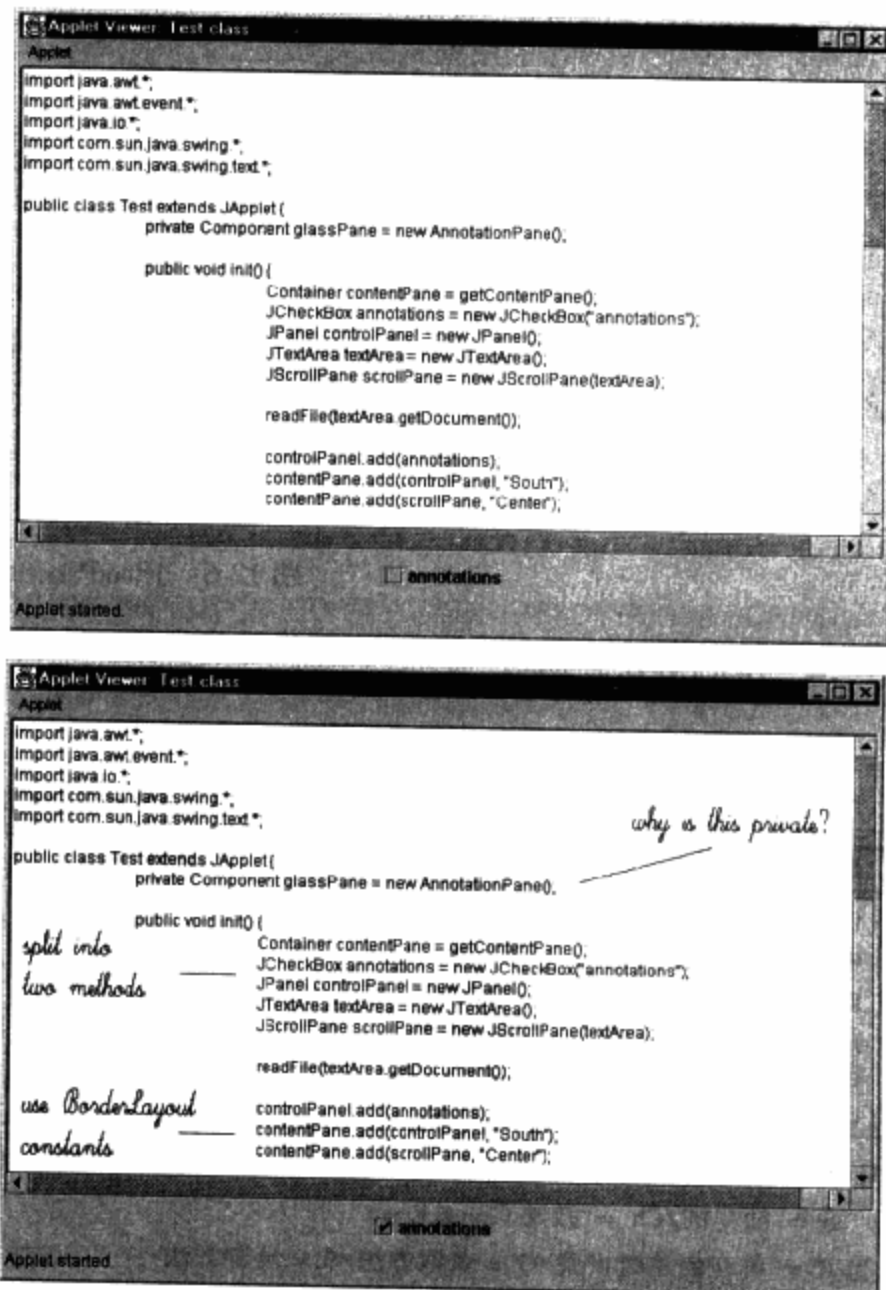


图 12-7 使用玻璃窗格来显示注释

```

public void init () {
    createContainerHierarchy ();
    setupGlassPane ();
}
...

```

`createContainerHierarchy` 方法创建了一个 `JPanel` 实例来容纳复选框。创建一个 `JTextArea` 实例以便显示文档，文档的文本区是嵌套在一个 `JScrollPane` 实例中的。把复选框添加到这个面板中，滚动窗格和面板则添加到这个小应用程序的内容窗格中。`readFile` 方法从源代码文件中读取文本并把文本添加到文本区中。

```

...
private void createContainerHierarchy () {
    Container contentPane = getContentPane ();

    JPanel controlPanel = new JPanel ();
    JTextArea textArea = new JTextArea ();
    JScrollPane scrollPane = new JScrollPane (textArea);

    readFile (textArea.getDocument ());

    controlPanel.add (annotations);

    contentPane.add (scrollPane, "Center"); // scroll pane
    contentPane.add (controlPanel, "South"); // panel

    textArea.addMouseListener (new MouseAdapter () {
        public void mouseEntered (MouseEvent e) {
            System.out.println ("enter");
        }
    });
}
...

```

`setupGlassPane` 方法调用 `JApplet.setGlassPane` 方法，这个方法进而调用这个小应用程序的根窗格的 `setGlassPane ()` 方法，正如 12.2.1 节“`RootPaneContainer` 接口”中所讨论的那样。不论何时选取或取消选取复选框，在复选框上添加的子项监听器都设置玻璃窗格的可见性。

```

...
private void setupGlassPane () {
    setGlassPane (glassPane);

    annotations.addItemListener (new ItemListener () {
        public void itemStateChanged (ItemEvent e) {
            if (e.getStateChange () == ItemEvent.SELECTED)
                glassPane.setVisible (true);
            else
                glassPane.setVisible (false);
        }
    });
}
...

```

例 12-4 中完整地列出了图 12-7 中示出的小应用程序的代码。

例 12-4 一个传递事件的玻璃窗格。

```

import java.awt.*;
import java.awt.event.*;
import java.io.*;

```



```

import javax.swing.*;
import javax.swing.text.*;

public class Test extends JApplet {
    Component glassPane = new AnnotationPane ();
    JCheckBox annotations = new JCheckBox ("annotations");

    public void init () {
        createContainerHierarchy ();
        setupGlassPane ();
    }

    private void createContainerHierarchy () {
        Container contentPane = getContentPane ();

        JPanel controlPanel = new JPanel ();
        JTextArea textArea = new JTextArea ();
        JScrollPane scrollPane = new JScrollPane (textArea);
        readFile (textArea.getDocument ());

        controlPanel.add (annotations);

        contentPane.add (scrollPane, "Center"); // scroll pane
        contentPane.add (controlPanel, "South"); // panel

        textArea.addMouseListener (new MouseAdapter () {
            public void mouseEntered (MouseEvent e) {
                System.out.println ("enter");
            }
        });
    }

    private void setupGlassPane () {
        setGlassPane (glassPane);

        annotations.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                if (e.getStateChange () == ItemEvent.SELECTED)
                    glassPane.setVisible (true);
                else
                    glassPane.setVisible (false);
            }
        });
    }

    private void readFile (Document doc) {
        try {
            Reader in = new FileReader ("Test.java");
            char [] buff = new char [4096];
            int next;

            while ( (next = in.read (buff, 0, buff.length)) != -1)
                doc.insertString (
                    doc.getLength (), new String (buff, 0, next), null);
        }
        catch (Exception e) {
            System.out.println ("interruption");
        }
    }
}

class AnnotationPane extends JPanel {
    private Icon annotations [] = {

```

```

        new ImageIcon ("annotation.gif"),
        new ImageIcon ("annotation_1.gif"),
        new ImageIcon ("annotation_2.gif")
    };
    public void paintComponent (Graphics g) {
        annotations [0] .paintIcon (this, g, 400, 50);
        annotations [1] .paintIcon (this, g, 10, 150);
        annotations [2] .paintIcon (this, g, 10, 265);
    }
}

```

12.2.6 JRootPane 类总结

类总结 12-2 中列出了 JRootPane 类的 public 和 protected 变量和方法。

类总结 12-2 JRootPane

扩展: JComponent

实现: javax.accessibility.Accessible

1. 构造方法

```
public JRootPane ()
```

JRootPane 仅提供了一个无参数的构造方法。这个构造方法实例化根窗格的内容窗格、玻璃窗格、分层窗格和布局管理器。这个构造方法还把双缓存设置为 true 并把根窗格的背景颜色设置为 UIManager.getColor ("control") 返回的值。

2. 方法

(1) 初始化

```
public void addNotify ()
```

```
public void removeNotify ()
```

```
protected Container createContentPane ()
```

```
protected Component createGlassPane ()
```

```
protected JLayeredPane createLayeredPane ()
```

```
protected LayoutManager createRootLayout ()
```

JRootPane 重载 protected addImpl 方法以确保玻璃窗格总是根窗格中的顶层组件。

上述 protected create... 方法创建根窗格的内容窗格、玻璃窗格、分层窗格和布局管理器。这些方法都是从 JRootPane 的无参数构造方法中调用的, 且都可以在扩展中重载, 如果缺省值不合适的话。注意, 为设置一个根窗格的玻璃窗格、内容窗格、分层窗格和布局管理器而扩展 JRootPane 不是必须的, 因为 JRootPane 为它们提供了设置方法。

(2) 内容窗格/玻璃窗格/分层窗格/菜单条/缺省按钮的访问方法

```
public Container getContentPane ()
```

```
public Component getGlassPane ()
```

```
public JLayeredPane getLayeredPane ()
```

```
public LayoutManager getRootLayout ()
```

```
public JMenuBar getMenuBar ()
```

```
public void setContentPane (Container)
```

```
public void setGlassPane (Component)
```

```
public void setLayeredPane (JLayeredPane)
```

```
public void setRootLayout (LayoutManager)
```

```
public void setMenuBar (JMenuBar)
```

上面列出的这些方法是 JRootPane 属性的访问方法。分别为一个根窗格的内容窗格、缺省按钮、玻璃窗格、分层窗格和菜单栏提供了设置方法和获取方法。

(3) 焦点循环和根验证

```
public boolean isFocusCycleRoot ()
public boolean isValidateRoot ()
```

JRootPane 重载 isFocusCycleRoot 以返回 true，即在根窗格中按下 Tab 键不把焦点移出根窗格。重载 isValidateRoot 并返回 true，意指重新绘制包含在一个根窗格中的组件，如果为根窗格中的一个组件调用了 revalidate() 方法的话。有关 revalidate() 方法和 isValidateRoot() 方法的详细内容参见 4.3.5 节。

(4) 其它方法

```
protected void addImpl (Component, Object, int)
public AccessibleContext getAccessibleContext ()
protected String paramString ()
```

JRootPane 实现 Accessible 接口，因此，提供了一个 getAccessibleContext 方法。

12.2.7 AWT 兼容

AWT 没有提供一个与 JRootPane 类似的组件。

12.3 JLayeredPane

JLayeredPane 是一个把组件放在不同的层上的组件。层可以用来控制组件的深度，它比 AWT 提供的缺省层序机制更精确。有关 AWT 容器定义层序的方式的详细内容参见 2.3.1 节“层序”。表 12-3 中总结了 JLayeredPane 类定义的 6 种层。

表 12-3 JLayeredPane 定义的层

层	值	描述
FRAME_CONTENT_LAYER	-3000	最底下的层，根窗格的菜单条和内容窗格就是放在这个层中的
DEFAULT_LAYER	0	在窗体内容层之上的层。缺省时，组件就放在这个层中的
PALETTE_LAYER	100	缺省层上面的层。用于调色板和浮动工具条
MODAL_LAYER	200	调色板层上面的层。被模态对话框用于确保对话框出现在组件、调色板和浮动工具条
POPUP_LAYER	300	模态层上面的层。被弹出式用于确保弹出式菜单出现在组件、调色板和浮动工具条之上
DRAG_LAYER	400	最顶层。可用于拖动组件。或用于一个组件必须出现在所有其他组件的情况中

每个层都是用一个数值来指定的；具有较大值的层在值小的层之上显示。还可以用数值指定表 12-3 之外的其他层。例如，把一个组件的层设置为一个 1 到 99 之间的值将把这个组件放在缺省层和调色板层之间。

JLayeredPane 实例维护的层是一种逻辑结构；Swing 没有定义 Layer 类。层是通过 AWT 提供的层序机制实现的。除了层外，还可以通过指定一个组件相对于与同一层中其他组件的位置来指定这个组件在该层上的深度。

概括一下，JLayeredPane 实例中的每一个组件都维护三个属性。表 12-4 中对这些属性作了总结。

表 12-4 分层窗格组件的逻辑属性

属性	描述
Index	一个分层窗格维护的组件数组的索引。一个组件的索引与它的层序之间有直接的相互关系：带有较小索引值的组件在具有较大索引值的组件之上显示
Layer	一个组件所处的层。处于编号较低的层上的组件在处于编号较高的层上的组件之下显示
Position	一个组件相对于同一层中其他组件的位置。具有较低位置值的组件在具有较高位置值的之上显示

具有较低索引值的组件显示在具有较高索引值的组件之上。例如，一个索引值为 0 的组件显示在一个索引值为 1 的组件之上。

同样，具有较低位置值的组件显示在具有较高位置值的组件之上，假设这些组件都同处同一层的话。例如，如果两个组件处在同一层上，且一个组件的位置值为 0，另一个组件的位置值为 1，那么前一个组件将在后一个组件之上显示。

层所起作用的方式与索引和位置所起作用的方式正好相反。例如，处在 0 层上的组件显示在处在 1 层的组件之下。

由于层是用 AWT 提供的层序机制实现的，因此，回顾一下层序影响轻量 Swing 组件的方式是有用的。

Swing 提示

JLayeredPane 仅用于轻量组件

正如 2.3 节“混合使用 Swing 和 AWT 组件”中所述，重量组件总是显示在轻量组件之上。因此，把重量组件加入到分层窗格中是没有意义的。例如，如果把一个 java.awt.Button 实例添加到一个分层窗格中，那么不管它分配在哪个层中，它都将显示在所有驻留在这个分层窗格中的所有轻量组件之上。扩展 java.awt.Component 的对象能够添加到分层窗格中，并且可为它们分配层。但是，这并不意味着把扩展 java.awt.Component 的对象添加到一个分层窗格中是有意义的。只有扩展 java.awt.Component 的轻量对象才应当添加到一个分层窗格中。

12.3.1 回顾轻量组件的层序

图 12-8 中示出的小应用程序包含了 6 个 JButton 实例，它们的位置和大小都显式地设置了。每一个组件的层序，也即深度是由这些按钮添加到这个小应用程序内容窗格中的顺序控制的。添加到内容窗格中的第一个按钮在其他按钮之上显示，而添加到内容窗格中的最后一个按钮在所有其他按钮之下显示。

例 12-5 中列出了图 12-8 中示出的小应用程序的代码。

例 12-5 添加到一个内容窗格中的按钮的层序

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class Test extends JApplet {
    Container cp = getContentPane ();

    private final Component [] comps = {
        new JButton (), new JButton (),
        new JButton (), new JButton (),
        new JButton (), new JButton (),
    };

    public void init () {
        cp.setLayout (null);

        for (int i=0; i < comps.length; ++i) {
            AbstractButton button = (AbstractButton) comps [i];
            cp.add (button);

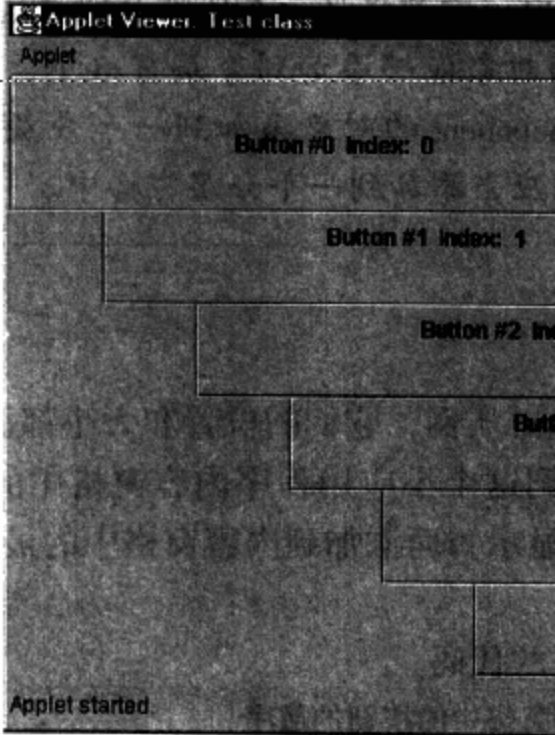
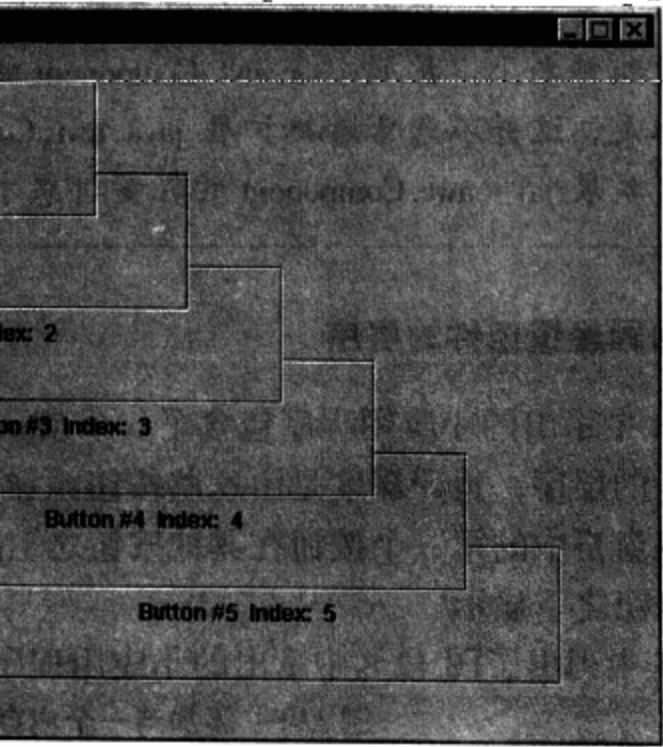
            String t = "Button #";
            t += i + " Index: " + getIndexOf (button);
            button.setText (t);
            button.setBounds (i * 50, i * 50, 350, 75);
            System.out.println ("Adding: " + button.getText ());
        }

        private int getIndexOf (Component button) {
            int ncomponents = cp.getComponentCount ();

            for (int i=0; i < ncomponents; ++i) {
                Component c = cp.getComponent (i);
```

```
                if (button == c)
                    return i;
            }

            return -1;
        }
    }
}
```



轻量组件的层序

处理器设置为 null，并在放置这些按钮时使它们一
加入内容窗格的顺序。JPanel 类没有提供一个方

图 12-8 Swing

这个小应用程序把它的内容窗格的布局管
个重叠另一个。每个按钮的文本指出了按钮被

法来返回一个组件的索引，因此这个小应用程序实现了一个 `getIndexOf` 方法来返回组件的索引。

这个小应用程序还在把按钮加入内容窗格时打印与按钮有关的信息。下面是这个小应用程序的输出：

```
Adding: Button #0 Index: 0
Adding: Button #1 Index: 1
Adding: Button #2 Index: 2
Adding: Button #3 Index: 3
Adding: Button #4 Index: 4
Adding: Button #5 Index: 5
```

12.3.2 为组件分配层

图 12-9 中示出的小应用程序与图 12-8 中示出的小应用程序类似，不同之处在于这个小应用程序的内容窗格被设置为一个 `JLayeredPane` 实例。每个按钮都用 `JLayeredPane.setLayer` 方法分配了一个特定的层。

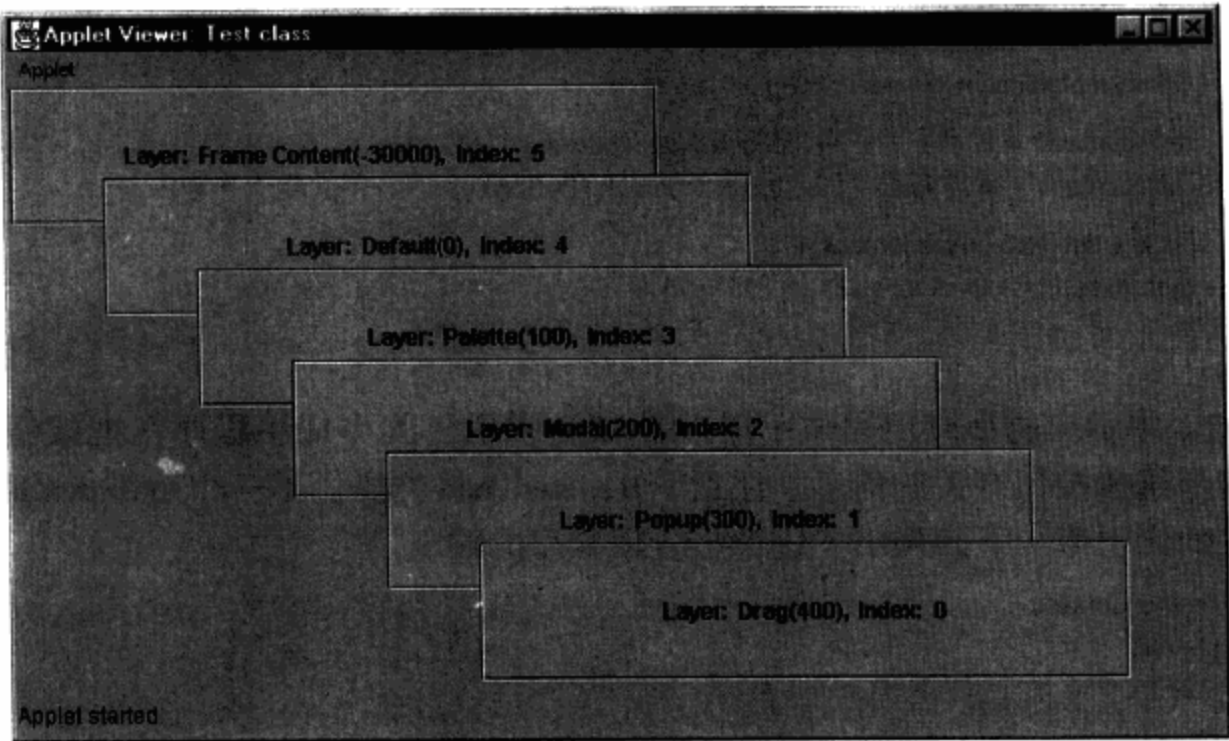


图 12-9 为一个分层窗格中组件分配层

这个小应用程序用表 12-3 中列出的 `JLayeredPane` 常数定义了一个 `Integer` 数组。这些 `Integer` 值用于在把这些按钮添加到分层窗格中之前设置每个按钮的层。

```
public class Test extends JApplet {
    private JLayeredPane lp = new JLayeredPane ();
    private Integer [] layers = {
        JLayeredPane.FRAME_CONTENT_LAYER,
        JLayeredPane.DEFAULT_LAYER,
        JLayeredPane.PALETTE_LAYER,
        JLayeredPane.MODAL_LAYER,
        JLayeredPane.POPUP_LAYER,
        JLayeredPane.DRAG_LAYER,
    };
    private final Component [] comps = {
        new JButton ("Frame Content"), new JButton ("Default"),
        new JButton ("Palette"), new JButton ("Modal"),
    }
```

```

        new JButton ("Popup"), new JButton ("Drag"),
    };
    public void init () {
        ...
        for (int i=0; i < comps.length; ++i) {
            AbstractButton button = (AbstractButton) comps [i];
            ...
            lp.setLayer (button, layers [i] .intValue ());
            lp.add (button);
        }
        ...
    }

```

在这些按钮添加到这个分层窗格中后，为这些按钮设置了文本以反映每个按钮所处的层及每个按钮的索引值。注意，代表层的 `JLayeredPane` 常数是 `Integer` 值；但是，`JLayeredPane.setLayer()` 带入的是一个 `int` 值。由于这种情况，使用了 `Integer.intValue()` 方法把 `Integer` 值转换为 `int` 值。

```

    ...
    for (int i=0; i < comps.length; ++i) {
        AbstractButton button = (AbstractButton) comps [i];
        String t = button.getText ();
        String replacement = new String ("Layer: ");
        replacement += t + " (" + lp.getLayer (button) + ")," ;
        replacement += " Index: " + lp.getIndexOf (button) ;
        button.setText (replacement);
        button.setBounds (i*50, i*50, 350, 75);
    }
}

```

与图 12-8 中示出的小应用程序一样，图 12-9 中示出的小应用程序在把按钮添加到分层窗格中时打印与这些按钮有关的信息。注意，`JLayeredPane` 类提供了一个 `getIndexOf` 方法来返回一个特定组件的索引值。下面列出了这个小应用程序的输出。

```

Adding: Frame Content
Adding: Default
Adding: Palette
Adding: Modal
Adding: Popup
Adding: Drag

```

关于这个小应用程序与图 12-8 中示出的小应用程序的对比有两个要点。

首先，由于这个小应用程序使用了一个 `JLayeredPane` 实例，因此，这些按钮加入到分层窗格中的顺序不像每个按钮的层序那么重要。第一个添加到分层窗格中按钮不是在其他按钮之上显示的，这与图 12-8 中示出的小应用程序中情况不同。事实上，第一个添加到分层窗格中的按钮被分配到 `FRAME_CONTENT_LAYER` 层中，根据定义，这个层是最底部的层。

其次，每个按钮的文本不是在把按钮添加到分层窗格中的 `for` 循环中设置的，因为在把所有的按钮添加到分层窗格中之前，每个按钮的索引值是不确定的。在把其他按钮添加到分层窗格中时，`JLayeredPane` 实例根据每个组件的层和位置不断地调整其所包含的组件的索引值。例如，在把第一个按钮添加到分层窗格中后，这个按钮的索引值是 0，当从图 12-9 中可以看出，在所有的按钮都添加到分层窗格中后，第一个按钮的索引值是 5。

例 12-6 中列出了图 12-9 中示出的小应用程序的代码。

例 12-6 把包含在一个分层窗格中的组件分配到层中

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JLayeredPane lp = new JLayeredPane ();

    private Integer [] layers = {
        JLayeredPane.FRAME_CONTENT_LAYER,
        JLayeredPane.DEFAULT_LAYER,
        JLayeredPane.PALETTE_LAYER,
        JLayeredPane.MODAL_LAYER,
        JLayeredPane.POPUP_LAYER,
        JLayeredPane.DRAG_LAYER,
    };

    private final Component [] comps = {
        new JButton ("Frame Content"), new JButton ("Default"),
        new JButton ("Palette"), new JButton ("Modal"),
        new JButton ("Popup"), new JButton ("Drag"),
    };

    public void init () {
        setContentPane (lp);
        lp.setLayout (null);

        for (int i=0; i < comps.length; ++i) {
            AbstractButton button = (AbstractButton) comps [i];

            System.out.println ("Adding: " + button.getText ());

            lp.setLayer (button, layers [i].intValue ());
            lp.add (button);
        }

        for (int i=0; i < comps.length; ++i) {
            AbstractButton button = (AbstractButton) comps [i];
            String t = button.getText ();
            String replacement = new String ("Layer: ");

            replacement += t + " (" + lp.getLayer (button) + ")," +
                replacement + " Index: " + lp.indexOf (button);

            button.setText (replacement);
            button.setBounds (i*50, i*50, 350, 75);
        }
    }
}

```

12.3.3 指定同一层中组件的位置

一个组件相对于同层中其他组件的位置可以通过指定这个组件的位置来控制。JLayeredPane 为这个目的提供了一个 setPosition 方法。

图 12-10 中示出的小应用程序图示了在同一层中放置组件的缺省方式。

这个小应用程序在缺省层中包含了三个按钮。缺省情况下，驻留在同一层上的组件的位置是由它们添加到分层窗格中的顺序决定的。在图 12-10 示出的小应用程序中，驻留在缺省层中

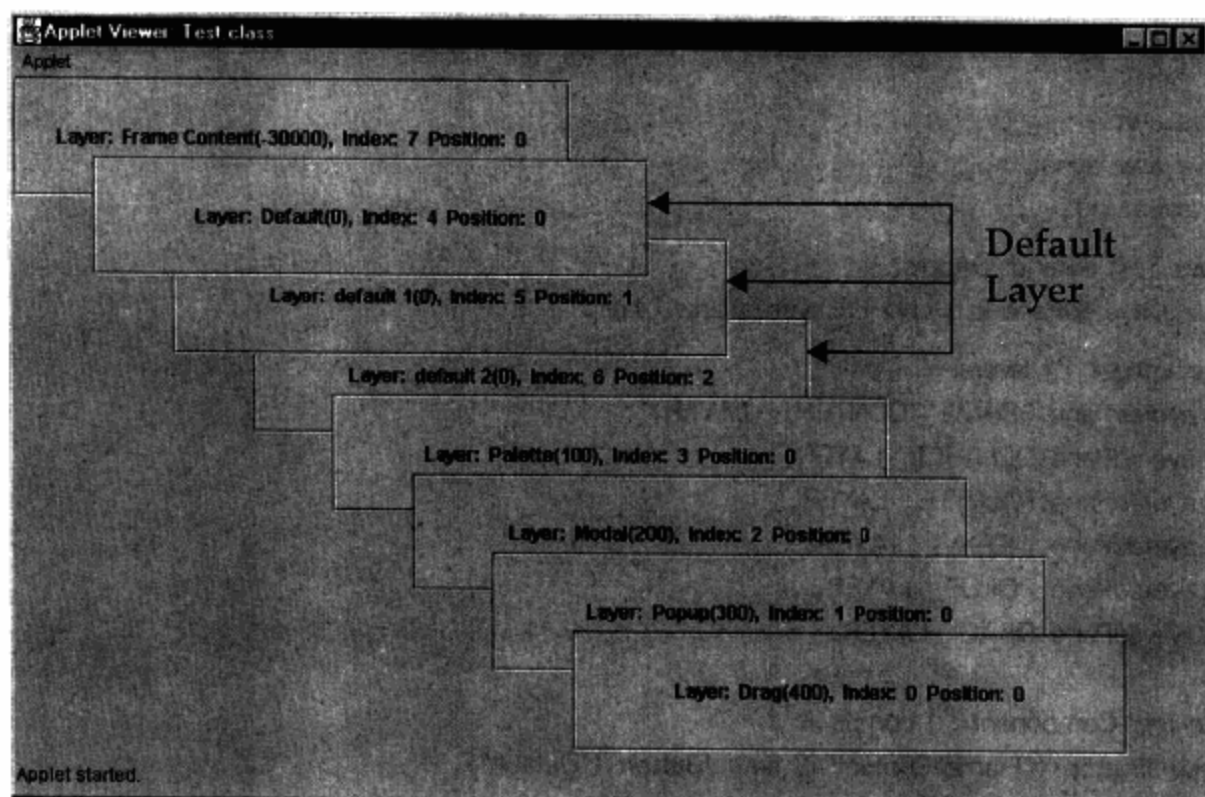


图 12-10 在同一层中放置组件的缺省方式

的按钮是以这样的顺序添加到分层窗格中的：“Default”、“default1”和“default2”。因此，“Default”按钮显示在“default1”按钮之上，而后者又显示在“default2”按钮之上。

例 12-7 列出了图 12-10 中示出的小应用程序的代码。

例 12-7 同一层中放置组件的缺省方式

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JLayeredPane lp = new JLayeredPane ();

    private Integer [] layers = {
        JLayeredPane.FRAME_CONTENT_LAYER,
        JLayeredPane.DEFAULT_LAYER,
        JLayeredPane.DEFAULT_LAYER,
        JLayeredPane.DEFAULT_LAYER,
        JLayeredPane.PALETTE_LAYER,
        JLayeredPane.MODAL_LAYER,
        JLayeredPane.POPUP_LAYER,
        JLayeredPane.DRAG_LAYER,
    };

    private final Component [] comps = {
        new JButton ("Frame Content"), new JButton ("Default"),
        new JButton ("default 1"), new JButton ("default 2"),
        new JButton ("Palette"), new JButton ("Modal"),
        new JButton ("Popup"), new JButton ("Drag"),
    };

    public void init () {
        setContentPane (lp);
        lp.setLayout (null);

        for (int i=0; i < comps.length; ++i) {
```

```

AbstractButton button = (AbstractButton) comps [i];
lp.setLayer (button, layers [i] .intValue ());
lp.add (button);
}
for (int i=0; i < comps.length; ++i) {
    AbstractButton button = (AbstractButton) comps [i];
    String t = button.getText ();
    String replacement = new String ("Layer: ");
    replacement += t + " (" + lp.getLayer (button) + ").";
    replacement += " Index: " + lp.indexOf (button);
    replacement += " Position: " +
        lp.getPosition (button);
    button.setText (replacement);
    button.setBounds (i * 50, i * 50, 350, 75);
}
}
}

```

图 12-11 中示出的小应用程序与图 12-10 示出的小应用程序几乎完全相同，不同之处在于它显式地设置了缺省层上的三个按钮的位置。

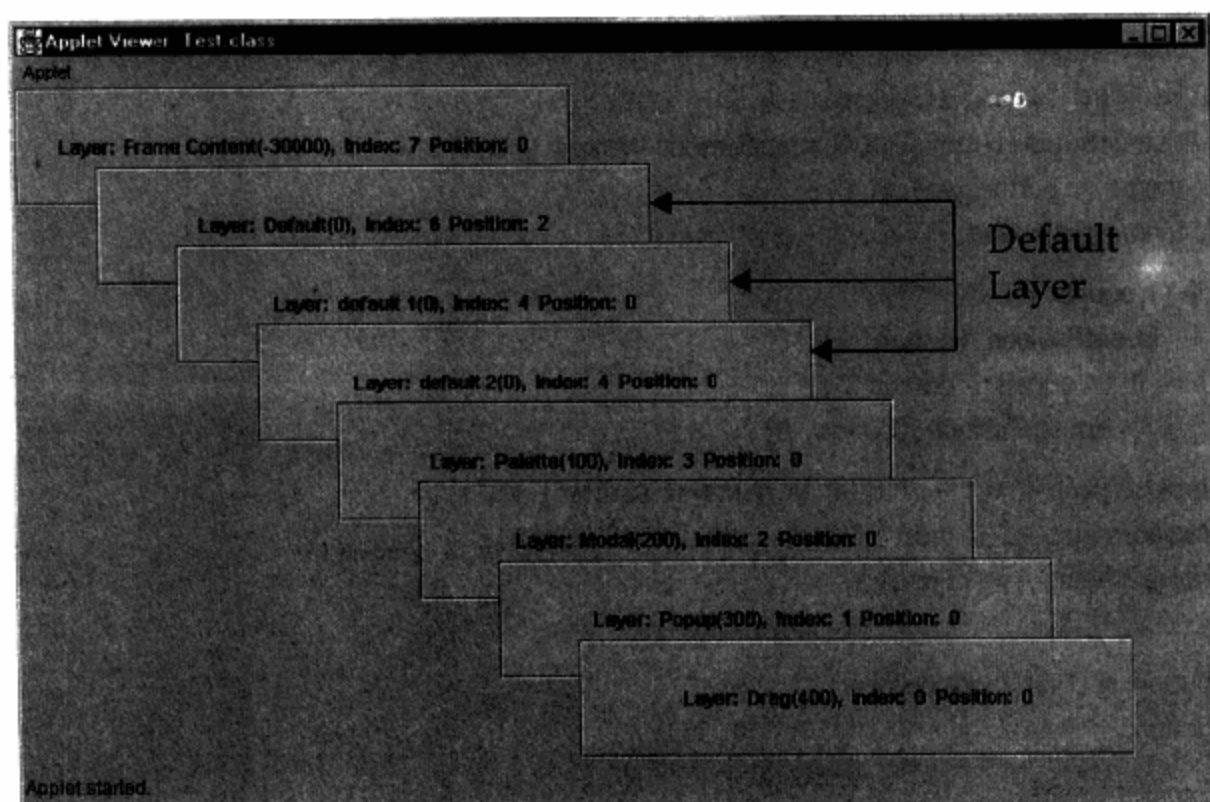


图 12-11 显式地设置同层中组件的位置

例 12-8 中列出了图 12-11 中示出的小应用程序的代码。

例 12-8 显式地设置同层中组件的位置

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JLayeredPane lp = new JLayeredPane ();
    private Integer [] layers = {
        JLayeredPane.FRAME_CONTENT_LAYER,

```

```

JLayeredPane.DEFAULT_LAYER,
JLayeredPane.DEFAULT_LAYER,
JLayeredPane.DEFAULT_LAYER,
JLayeredPane.PALETTE_LAYER,
JLayeredPane.MODAL_LAYER,
JLayeredPane.POPUP_LAYER,
JLayeredPane.DRAG_LAYER,
};
private final Component [] comps = {
    new JButton ("Frame Content"), new JButton ("Default"),
    new JButton ("default 1"), new JButton ("default 2"),
    new JButton ("Palette"), new JButton ("Modal"),
    new JButton ("Popup"), new JButton ("Drag"),
};
public void init () {
    setContentPane (lp);
    lp.setLayout (null);

    for (int i=0; i < comps.length; ++i) {
        AbstractButton button = (AbstractButton) comps [i];
        String t = button.getText ();

        lp.setLayer (button, layers [i].intValue ());
        lp.add (button);
    }

    for (int i=0; i < comps.length; ++i) {
        AbstractButton button = (AbstractButton) comps [i];
        String t = button.getText ();
        String replacement = new String ("Layer: ");

        if (t.equals ("Default"))
            lp.setPosition (button, 2);
        else if (t.equals ("default 2"))
            lp.setPosition (button, 0);

        replacement += t + " (" + lp.getLayer (button) + ")," +
        replacement += " Index: " + lp.indexOf (button);
        replacement += " Position: " +
            lp.getPosition (button);

        button.setText (replacement);
        button.setBounds (i*50, i*50, 350, 75);
    }
}

```

例 12-8 中列出的小应用程序通过显式地设置 “Default” 和 “default2” 的位置来交换它们的深度。注意，直到把所有的按钮都添加到分层窗格中之后才指定按钮的位置，因为与索引值一样，随着组件添加到分层窗格中，组件位置也是不断地调整的。

Swing 提示

层是以 int 值指定的

`JLayeredPane.setLayer()` 方法带入了一个组件和一个 int 值。这个 int 值指定这个组件放在哪个层上。另一方面，用于描述层的 `JLayeredPane` 常数，如 `JLayeredPane.DEFAULT_LAYER`

和 `JLayeredPane.POPUP_LAYER` 等是作为 `Integer` 值定义的。如果一个组件将放置到一个用表 12-3 中的一个 `JLayeredPane` 常数定义的层中，那么，这个 `integer` 值必须转换为一个 `int` 值。这可以通过调用 `Integer.intValue` 方法来实现，如下所示：

```
aLayeredPane.setLayer (aComponent, JLayeredPane.POPUP_LAYER.intValue ());
```

12.3.4 使用拖动层

由于拖动层处在所有其他层之上，因此，它可以用于在一个分层窗格包含的所有其他组件之上拖动组件。通过拖动一个 `JLabel` 实例，图 12-12 显示了一个实际的拖动层。这个小应用程序包含两个选项卡和复选框。吸烟的骷髅像的选项卡能够拖动，拳头图片的选项卡则是静态的。复选框控制可拖动选项卡是否驻留在拖动层中。如果选取了复选框，则把可拖动选项卡的层设置为拖动层；如果未选取复选框，则把可拖动选项卡的层设置为缺省层。

图 12-12 中左上图示出了这个小应用程序开始时的样子。两个选项卡在添加到分层窗格中时都没有指定层，因此，两个选项卡都放置在缺省层上。

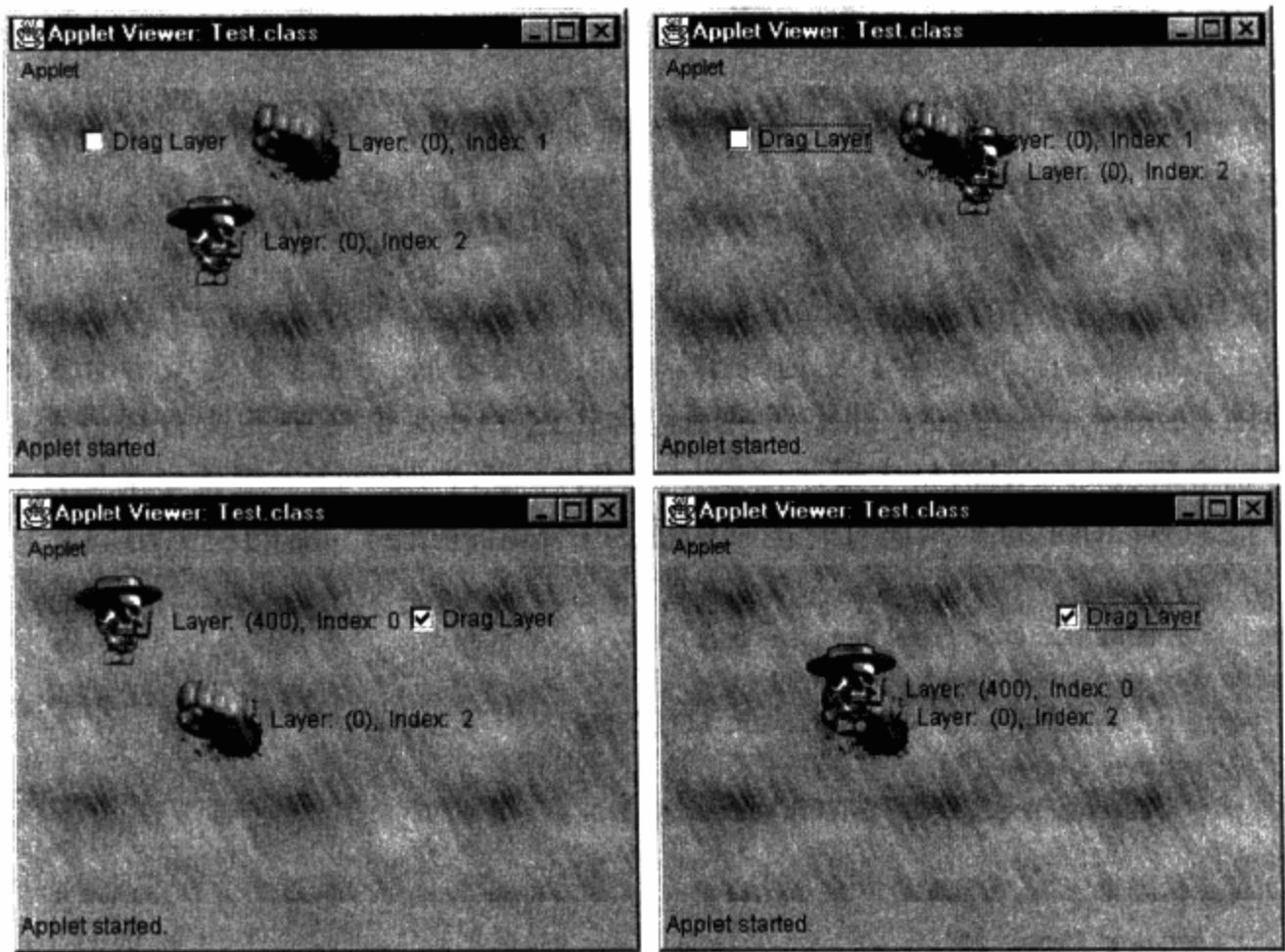


图 12-12 在一个分层窗格中使用拖动层

这个小应用程序把它的内容窗格设置为一个扩展 `JLayeredPane` 的 `CustomContentPane` 实例。`CustomContentPane` 实例化图标、选项卡和复选框，并在不指定层的情况下添加它们。注意，`CustomContentPane` 把它的布局管理器设置为一个 `FlowLayout` 实例，因为，缺省情况下，`JLayeredPane` 具有一个 `null` 布局管理器。

```
public class Test extends JApplet {
    public void init () {
        setContentPane (new CustomContentPane ());
    }
}
```

```

class CustomContentPane extends JLayeredPane {
    private ImageIcon rain = new ImageIcon ("rain.gif");
    private ImageIcon punch = new ImageIcon ("punch.gif");
    private ImageIcon skelly = new ImageIcon ("skelly.gif");
    private int rainw = rain.getIconWidth ();
    private int rainh = rain.getIconHeight ();

    private JLabel [] labels = {
        new JLabel ("I stay put", punch, SwingConstants.RIGHT),
        new JLabel ("Drag me around!",
            skelly, SwingConstants.RIGHT),
    };

    public CustomContentPane () {
        Dragger listener = new Dragger ();
        JCheckBox onDragLayer = new JCheckBox ("Drag Layer");

        // JLayeredPane has a null layout by default
        setLayout (new FlowLayout ());

        add (onDragLayer);
        add (labels [0]);
        add (labels [1]);
        ...
    }
}

```

图 12-12 中右上图示出了这个小应用程序在拖动可拖动选项卡后的样子。由于拳头选项卡是第一个添加到分层窗格中的选项卡，因此，它显示在可拖动选项卡的上面，所以可拖动选项卡是在拳头选项卡下面拖动的。

图 12-12 中左下图示出了这个小应用程序在选取复选框后的样子。一个为可拖动选项卡设置相应层的子项监听器添加到这个复选框上。在设置层之后，与这个选项卡有关的文本被更新，并且验证 CustomContentPane，即重新绘制包含在这个分层窗格中的所有组件都。注意，这个可拖动选项卡现在是分层窗格中显示的第一个组件。因为改变这个选项卡的层还改变了它的索引值，并且 FlowLayout 的实例缺省时把索引值为 0 的组件放到容器的左上角。

```

...
onDragLayer.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        if (e.getStateChange () == ItemEvent.SELECTED) {
            setLayer (labels [1],
                JLayeredPane.DRAG_LAYER.intValue ());
        }
        else {
            setLayer (labels [1],
                JLayeredPane.DEFAULT_LAYER.intValue ());
        }
        setLabelText ();
        validate ();
    }
});

private void setLabelText () {
    for (int i=0; i < labels.length; ++i) {
        JLabel label = labels [i];
        String t = new String ("Layer: ");
        t += " (" + getLayer (label) + ")," ;
    }
}

```

```

        t += " Index: " + getIndexOf (label);
        label.setText (t);
    }
    ...
}

```

图 12-12 中右下图示出了在可拖动选项卡放置到拖动层并被拖动后的样子。由于可拖动选项卡是放在拖动层上的，因此，它在静态选项卡之上显示。

例 12-9 完整地列出了图 12-12 中示出的小应用程序的代码。

例 12-9 使用拖动层

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
    public void init () {
        setContentPane (new CustomContentPane ());
    }
}

class CustomContentPane extends JLayeredPane {
    private ImageIcon rain = new ImageIcon ("rain.gif");
    private ImageIcon punch = new ImageIcon ("punch.gif");
    private ImageIcon skelly = new ImageIcon ("skelly.gif");
    private int rainw = rain.getIconWidth ();
    private int rainh = rain.getIconHeight ();

    private JLabel [] labels = {
        new JLabel ("I stay put", punch, SwingConstants.RIGHT),
        new JLabel ("Drag me around!",
                    skelly, SwingConstants.RIGHT),
    };

    public CustomContentPane () {
        Dragger listener = new Dragger ();
        JCheckBox onDragLayer = new JCheckBox ("Drag Layer");

        // JLayeredPane has a null layout by default
        setLayout (new FlowLayout ());

        add (onDragLayer);
        add (labels [0]);
        add (labels [1]);

        labels [1].addMouseMotionListener (listener);
        labels [1].addMouseListener (listener);

        setLabelText ();

        onDragLayer.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                if (e.getStateChange () == ItemEvent.SELECTED) {
                    setLayer (labels [1],
                               JLayeredPane.DRAG_LAYER.intValue ());
                }
                else {
                    setLayer (labels [1],
                               JLayeredPane.DEFAULT_LAYER.intValue ());
                }
            }
        });
    }
}

```



```

        JLayeredPane.DEFAULT_LAYER.intValue ());
    }
    setLabelText ();
    validate ();
}
});
}
public void paintComponent (Graphics g) {
    Dimension size = getSize ();
    for (int row=0; row < size.height; row += rainh)
        for (int col=0; col < size.width; col += rainw)
            rain.paintIcon (this, g, col, row);
}
private void setLabelText () {
    for (int i=0; i < labels.length; ++i) {
        JLabel label = labels [i];
        String t = new String ("Layer: ");
        t += " (" + getLayer (label) + "),";
        t += " Index: " + getIndexOf (label);
        label.setText (t);
    }
}
}
class Dragger extends MouseAdapter
    implements MouseMotionListener {
    Point press = new Point ();
    boolean dragging = false;
    public void mousePressed (MouseEvent event) {
        press.x = event.getX ();
        press.y = event.getY ();
        dragging = true;
    }
    public boolean isDragging () {
        return dragging;
    }
    public void mouseReleased (MouseEvent event) {
        dragging = false;
    }
    public void mouseClicked (MouseEvent event) {
        dragging = false;
    }
    public void mouseMoved (MouseEvent event) {
        // don't care
    }
    public void mouseDragged (MouseEvent event) {
        Component c = (Component) event.getSource ();
        if (dragging) {
            Point loc = c.getLocation ();
            Point pt = new Point ();
            pt.x = event.getX () + loc.x - press.x;
            pt.y = event.getY () + loc.y - press.y;
            c.setLocation (pt.x, pt.y);
        }
    }
}

```

```
c.getParent().repaint();
```

组件总结 12-3 对 JLayeredPane 类进行了总结。

组件总结 12-3 JLayeredPane

- 模型：_____
- UI 代表：_____
- 绘制器：_____
- 编辑器：_____
- 激发的事件：_____
- 替代：_____
- 类图：_____

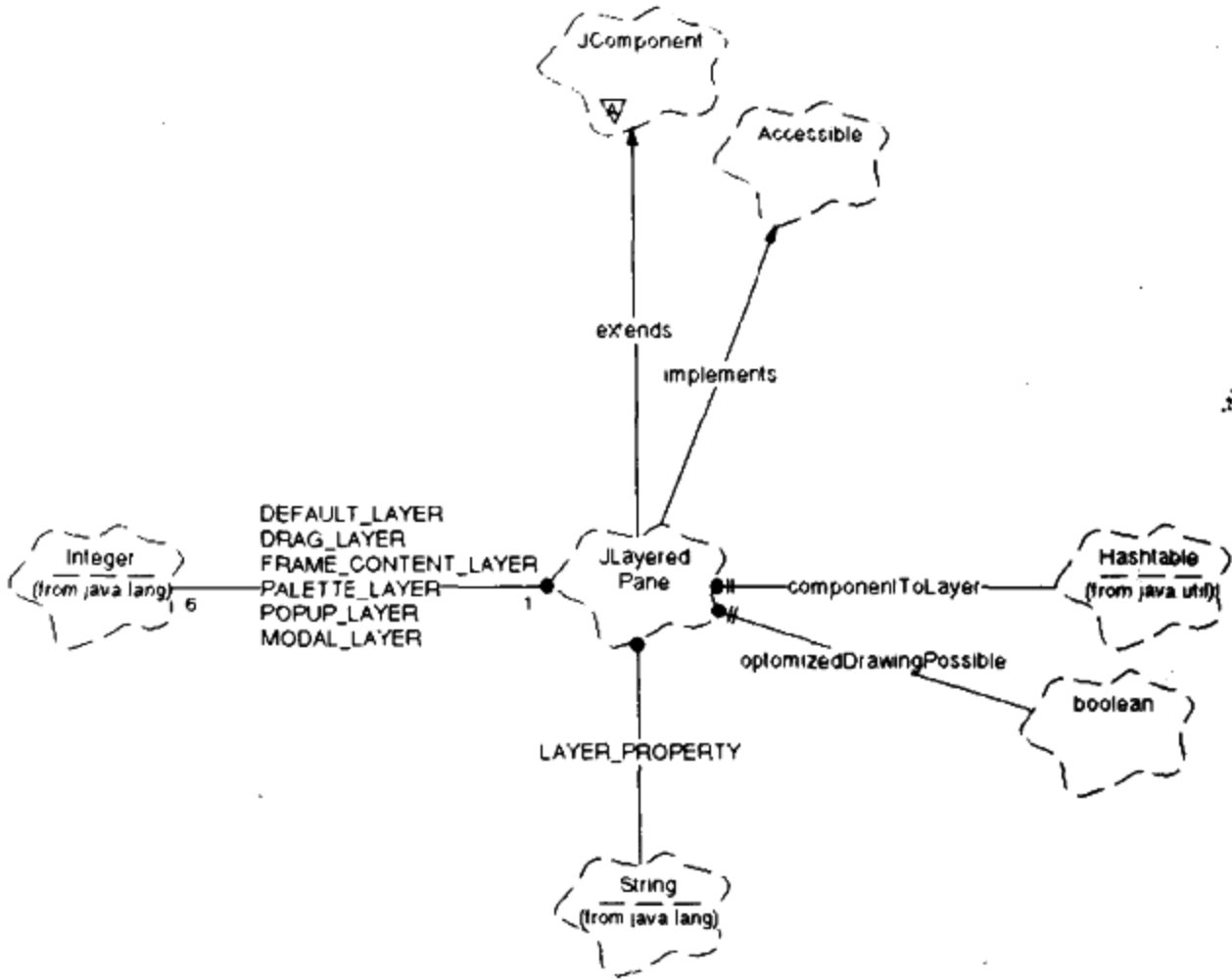


图 12-13 JLayeredPane 的类图

与 JRootPane 一样，JLayeredPane 是 JComponent 的一个简单扩展，没有模型或 UI 代表。JLayeredPane 以 Integer 的形式定义一些常数，这些常数定义了一组常用的层。另外，JLayeredPane 还为它的 layer 属性名定义了一个 String 常数。

JLayeredPane 还维护对一个 Hashtable 和一个 boolean 值的 protected 引用。Hashtable 跟踪包含在分层窗格和它的相应层中的每一个 java.awt.Component 实例。JComponent 实例的层是作为一个客户属性存储的。有关 JComponent 和客户属性的详细内容参见 4.9 节“客户属性”。

boolean 值记录是否可能对分层窗格采用优化绘制方法。如果在一个分层窗格中没有组件能够覆盖其他组件，那么优化绘制就是可行的，JLayeredPane 将以更有效的方式绘制它的组件。

12.3.5 JLayeredPane 属性

表 12-5 中列出了 JLayeredPane 维护的属性。

表 12-5 JLayeredPane 的属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
componentCountInLayer	int	I	G	—
componentInLayer	int	I	G	—
indexOf	int	S	G	—
optimizedDrawingEnabled	boolean	S	G	—
layer	int	S	SG	—
position	JRootPane	S	SG	—

① B = 关联的 (激发 PropertyChangeEvent) / C = 约束的 / I = 索引的 / S = 简单的 / Ch = 激发 ChangeEvent
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ L&F = 与界面样式有关

- componentCountInLayer**—— 包含在一个特定层中的组件数。
- componentInLayer**—— 包含在一个层中的一组组件。
- indexOf**—— 一个指定组件的索引。
- optimizedDrawingEnabled**—— 如果在一个分层窗格中没有组件重叠，则优化绘制是可行的。
- layer**—— 一个指定组件的层。
- position**—— 一个指定组件的位置。

JLayeredPane 的属性在 Swing 组件中有些特别，因为这些属性大多数是应用于包含在一个分层窗格中的组件的，而不是用于分层窗格本身的。只有 optimizedDrawingEnabled 属性应用于分层窗格本身；其余属性都应用于包含在分层窗格中的组件的。

注意 indexOf 属性是只读的，而 layer 和 position 属性是可以设置的。这是因为 indexOf 属性对应于分层窗格维护的组件数组的索引值，是由分层窗格在添加组件及设置组件的位置和层时控制的。

12.3.6 JLayeredPane 类总结

类总结 12-3 中列出了 JLayeredPane 类的 public 和 protected 变量和方法。

类总结 12-3 JLayeredPane

1. 构造方法

```
public JLayeredPane ()
```

JLayeredPane 提供了唯一一个没有参数的构造方法。这个构造方法把分层窗格的布局管理器设置为 null。

2. 方法

(1) 静态方法

```
public static int getLayer ()
public static void putLayer (JComponent, int)

public static JLayeredPane getLayeredPaneAbove (Component)
```

前两个方法具有等价的实例方法（非静态方法）：`setLayer (Component, int)` 和 `getLayer (Component)`。静态方法与实例方法之间的区别是：静态方法带入 `JComponent` 实例，而实例方法带入 `Component` 实例；静态方法不产生任何副作用。例如，`JLayeredPane.setLayer (Component, int)` 导致对一个设置了层的组件所占据的区域进行重绘，而静态方法 `JLayeredPane.putLayer (JComponent, int)` 不导致重绘操作。

`getLayeredPaneAbove (Component)` 返回包含这个方法带入的组件的分层窗格。如果传送给 `getLayeredPaneAbove ()` 方法的组件不在一个分层窗格中，则这个方法返回 `null`。

(2) 添加组件/组件与层

```
protected void addImpl (Component, Object constraints, int index)
protected int insertIndexForLayer (int, int)
protected Hashtable getComponentToLayer ()
```

重载保护的 `addImpl` 方法，以便根据组件的层和位置调整添加的组件的索引值。它的 `Object` 参数可以是一个指定组件的层的 `Integer` 值。如果 `JLayeredPane` 的扩展要在组件添加到分层窗格中时完成某种操作，则可以重载 `addImpl` 方法。实际应用中，极少重载这个方法。

`insertIndexForLayer` 方法根据一个组件的层和位置计算它的索引值。这个方法由 `addImpl` 方法调用。

`getComponentToLayer` 方法返回 `JLayeredPane` 的实例用于跟踪与 `java.awt.Component` 实例相关的层的 `Hashtable`。注意，这个 `Hashtable` 不必包含分层窗格中的所有组件，因为 `JComponent` 实例层是作为一个客户属性存储的。由于 `java.awt.Component` 实例没有客户属性，因此，用了另一种称作 `Hashtable` 的机制来记录层。

(3) 绘制/层访问

```
public void paint (Graphics)
public int getComponentCountInLayer (int layer)
public Component [] getComponentsInLayer (int layer)
public int highestLayer ()
public int lowestLayer ()
protected Integer getObjectForLayer (int layer)
public isOptimizedDrawingEnabled ()
```

`paint` 方法被 `JLayeredPane` 重载的原因与它被 `JPanel` 重载的原因相同。由于 `JLayeredPane` 和 `JPanel` 都没有一个 `UI` 代表，因此，这两个组件都重载 `paint` 方法，以使用背景色为不透明组件填充背景。如果没有为一个分层窗格显式地设置一个背景色，则将以亮灰色作为背景色。

上面列出的其余方法提供与分层窗格有关的信息。`getComponentCountInLayer` 方法返回一个特定层中包含的组件数，而 `getComponentsInLayer` 方法返回驻留在一个特定层中的组件的数组。`highestLayer` 方法和 `lowestLayer` 方法分别返回与最高层和最低层相关的数值。`getObjectForLayer` 方法根据一个表示层的 `int` 值返回一个 `Integer` 值。最后，`isOptimizedDrawingEnabled` 方法返回一个 `boolean` 值，这个值指示包含在分层窗格中的组件是否会重叠。如果没有两个组件会重叠，则 `JLayeredPane` 实例能够以一种效率更高的方式绘制它们的组件，并且这个方法返回 `true`。

(4) 索引/层/位置

```
public int getIndexOf (Component)
```

```
public int getLayer (Component)
public int getPosition (Component)
```

上面列出的方法返回与包含在一个分层窗格中的特定组件有关的信息。如果传送给这些方法的组件没有包含在分层窗格中，则 `getIndexOf` 和 `getPosition` 返回 -1，`getLayer` 返回 0。

(5) 设置层和位置/移动/删除组件

```
public void setLayer (Component, int layer)
public void setLayer (Component, int layer, int position)
public void setPosition (Component, int position)

public void moveToBack (Component)
public void moveToFront (Component)
public void remove (int)
```

上面方法对分层窗格中的一个特定组件进行操作。传送给第二个 `setLayer` 方法的两个 `int` 值分别指定组件的层和位置。`moveToFront` 和 `moveToBack` 方法调整组件在它的层中的位置。重载 `JComponent` 类的 `remove` 方法，以便在从分层窗格中删除一个组件时确定是否可以启用优化绘制方式。

(6) 可访问性/参数字符串

```
public AccessibleContext getAccessibleContext ()
protected String paramString ()
```

`JLayeredPane` 实现在 `Accessible` 接口中定义的 `getAccessibleContext` 方法。`paramString` 方法返回一个表示分层窗格的字符串。

12.3.7 AWT 兼容

AWT 没有一个与 `JLayeredPane` 类似的组件。

Swing 提示

缺省时，`JLayeredPane` 实例带有一个 `null` 布局管理器

如果在一个 `JLayeredPane` 实例中添加了一些组件，但这些组件没有在分层窗格中出现，那么可能是由于没有为分层窗格显式地设置布局管理器。缺省情况下，`JLayeredPane` 的实例带有一个 `null` 布局管理器，因此，添加到分层窗格中的组件不会绘制出来，除非为分层窗格显式地设置了布局管理器。

12.4 JTabbedPane

选项卡窗格是一种常用的用户界面组件，它们提供了方便地访问多个面板的途径。

Swing 的选项卡窗格是用 `JTabbedPane` 来实现的。包含在一个 `JTabbedPane` 实例中的选项卡有一个组件与它们相关联，这个组件在选项卡的下方显示。选项卡可以显示图标和文本，还可以设置颜色。另外，包含在一个选项卡窗格中的选项卡还可以与工具提示联系起来。

包含在一个 `JTabbedPane` 实例中的选项卡类似于一个 `JLayeredPane` 实例中的层，因为选项卡和层都是逻辑结构。即包含在 `JTabbedPane` 实例中的选项卡不是 Swing 组件。实际上，选项卡仅限于显示图标和文本，不能在选项卡上添加任意其他组件。

图 12-15 示出了带有一个选项卡窗格的小应用程序。这个选项卡窗格含有两个选项卡，它们分别与一个仅含有单个按钮的面板联系在一起。第二个选项卡显示了一个图标和一行文本，

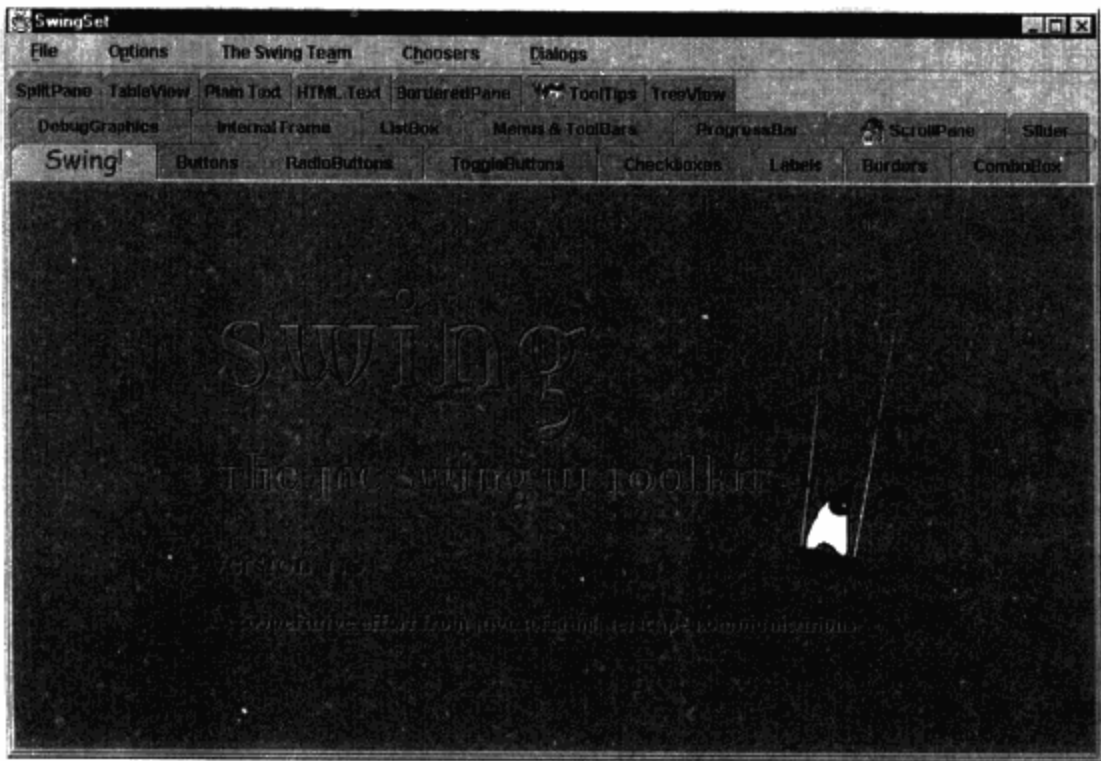


图 12-14 一个使用了选项卡窗格的例子 SwingSet

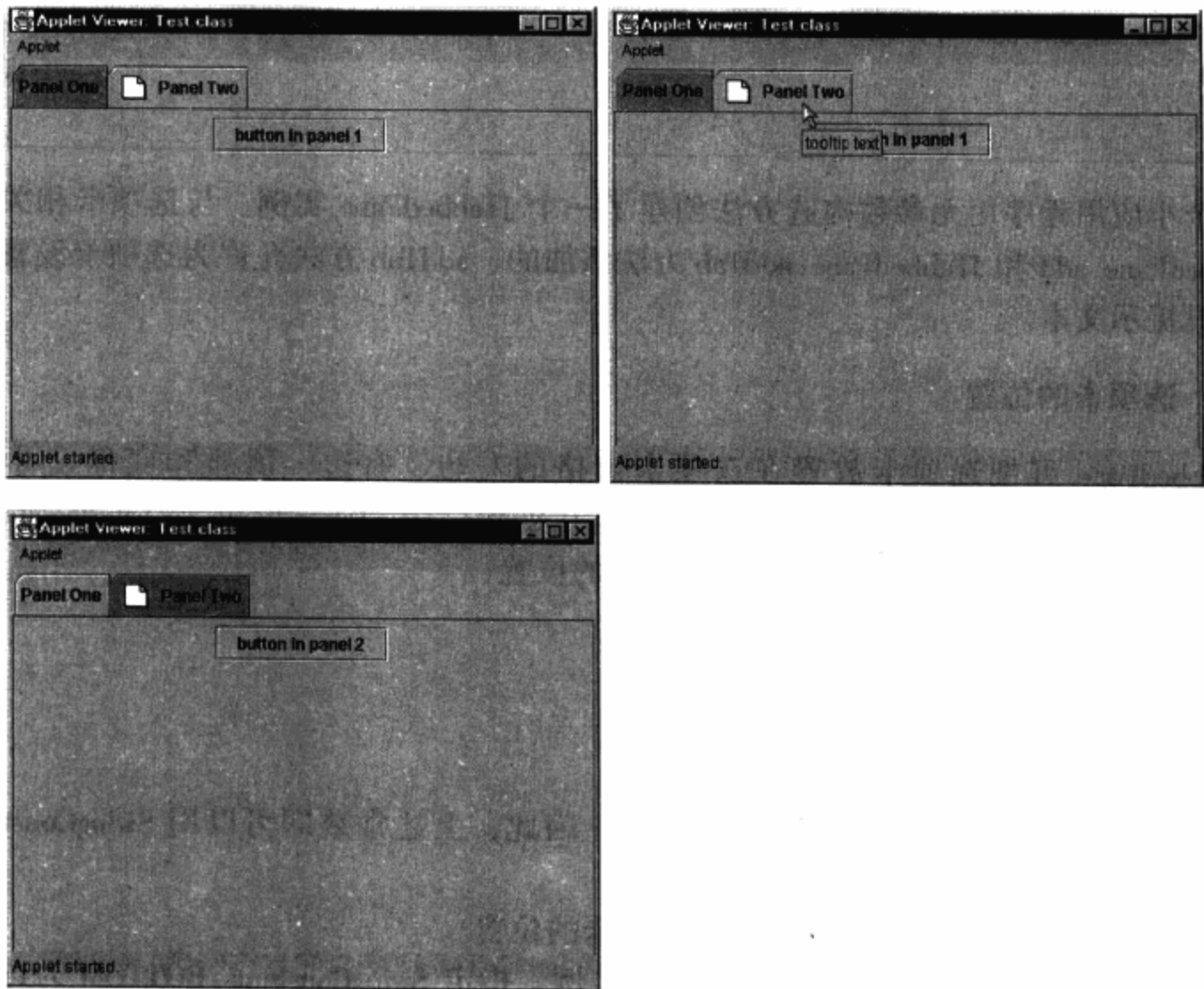


图 12-15 选项卡窗格

还有工具提示。

图 12-15 中左上图示出的是选取了“Panel One”选项卡时的样子。右上图示出的是选取了“Panel One”选项卡，但光标停留在“Panel Two”选项卡上时的样子。下图示出的是选取了“Panel Two”选项卡时的样子。

例 12-10 列出了图 12-15 中示出的小应用程序的代码。

例 12-10 JTabbedPane 的一个简单实例

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JTabbedPane tp = new JTabbedPane ();
        JPanel panelOne = new JPanel ();
        JPanel panelTwo = new JPanel ();

        panelOne.add (new JButton ("button in panel 1"));
        panelTwo.add (new JButton ("button in panel 2"));

        tp.add (panelOne, "Panel One");
        tp.addTab ("Panel Two",
            new ImageIcon ("document.gif"),
            panelTwo,
            "tooltip text");

        contentPane.add (tp, BorderLayout.CENTER);
    }
}
```

这个小应用程序用无参数构造方法创建了一个 JTabbedPane 实例。与选项卡相关的面板是用 JTabbedPane.add 和 JTabbedPane.addTab 方法添加的。addTab 方法允许为选项卡设置文本、图标和工具提示文本。

12.4.1 选项卡的位置

JTabbedPane 可把选项卡放置在选项卡窗格的左边、右边、顶部和底部。可以在创建 JTabbedPane 实例时指定选项卡的位置，也可以在创建后用 JTabbedPane.setTabPlacement 方法设置选项卡的位置。下述常数可用于指定选项卡的位置：

- SwingConstants.TOP
- SwingConstants.BOTTOM
- SwingConstants.LEFT
- SwingConstants.RIGHT

由于 JTabbedPane 实现 SwingConstants 接口，因此，上述常数即可以用 SwingConstants. 前缀指定，也可以用 JTabbedPane. 前缀指定。

图 12-16 中示出的小应用程序可设置选项卡的位置。

这个小应用程序创建了一个 JTabbedPane 实例，指定 SwingConstants.BOTTOM 为选项卡的初始位置。这个小应用程序还实例化了一个 JComboBox 实例，并为它提供了相应的字符串。

```
public class Test extends JApplet {
    private JTabbedPane tp =
        new JTabbedPane (SwingConstants.BOTTOM);
    private JComboBox combo = new JComboBox ();

    public Test () {
```



```

Container contentPane = getContentPane ();
JPanel comboPanel = new JPanel ();
JPanel panelOne = new JPanel ();
JPanel panelTwo = new JPanel ();

tp.add (panelOne, "Panel One");
tp.addTab ("Panel Two",
           new ImageIcon ("document.gif"),
           panelTwo,
           "tooltip text");

combo.addItem ("TOP");
combo.addItem ("LEFT");
combo.addItem ("RIGHT");
combo.addItem ("BOTTOM");
...

```

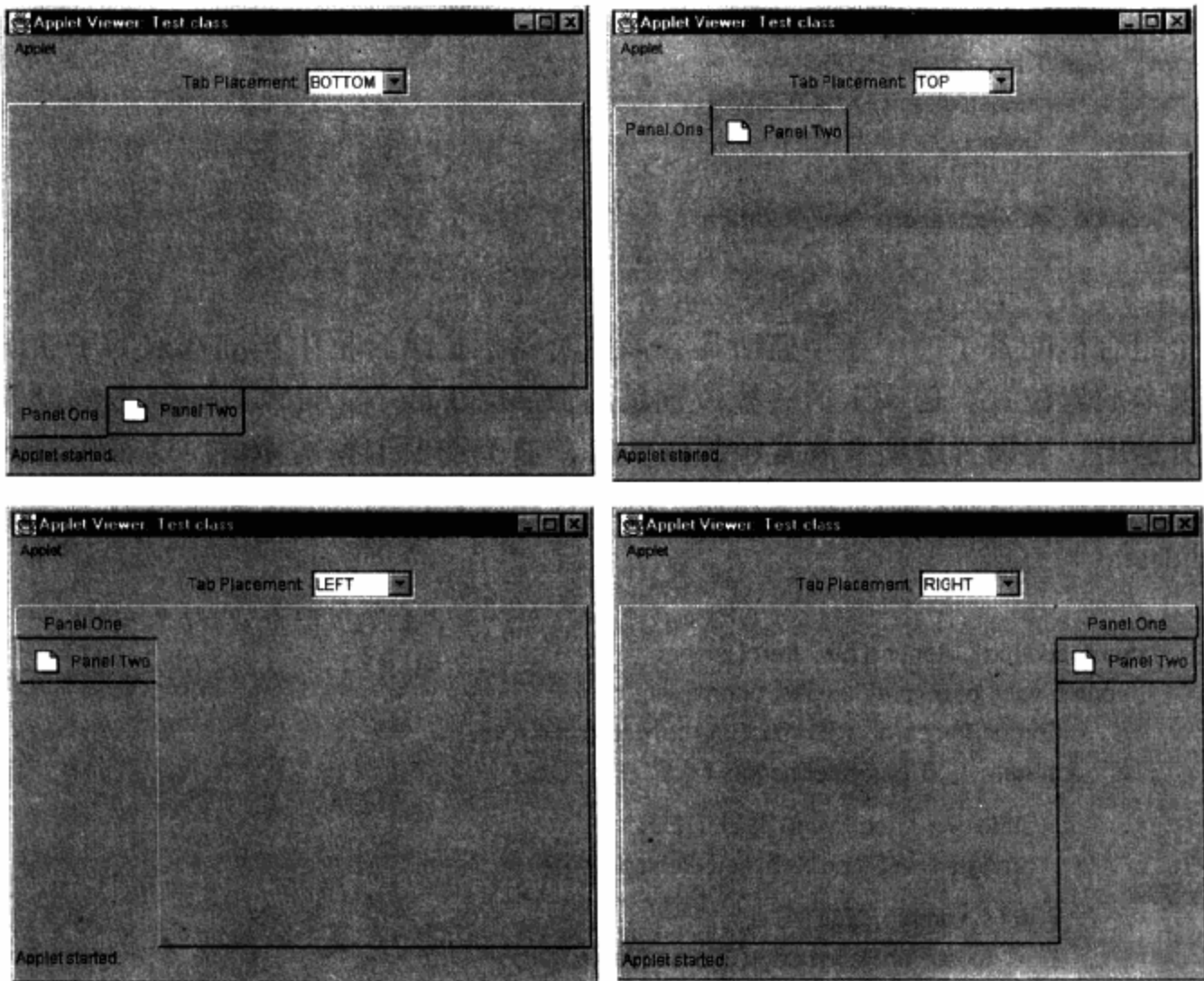


图 12-16 放置选项卡窗格的选项卡

把这个组合框和一个选项卡添加到一个面板中，把这个面板随后添加到这个小应用程序的内容窗格中的上部。然后，把一个分层窗格添加到这个小应用程序的内容窗格中的中部。接着调用了这个小应用程序实现的一个 `private` 方法——`setComboValue`，以便在组合框中选取子项。

`SetComboValue` 从 `JTabbedPane.getTabPlacement` 方法获得选项卡的初始位置并在组合框中选取相应的子项。

```

...
comboPanel.add (new JLabel ("Tab Placement:"));
comboPanel.add (combo);

contentPane.add (comboPanel, BorderLayout.NORTH);
contentPane.add (tp, BorderLayout.CENTER);

```

```

setComboValue ();
...
}
private void setComboValue () {
    int placement = tp.getTabPlacement ();
    String selectedItem = null;
    switch (placement) {
        case JTabbedPane.TOP:
            selectedItem = "TOP";
            break;
        case JTabbedPane.LEFT:
            selectedItem = "LEFT";
            break;
        case JTabbedPane.RIGHT:
            selectedItem = "RIGHT";
            break;
        case JTabbedPane.BOTTOM:
            selectedItem = "BOTTOM";
            break;
    }
    combo.setSelectedItem (selectedItem);
}
}

```

这个组合框配备了一个子项监听器，该监听器根据组合框中当前选取的子项设置选项卡窗格的选项卡放置位置。选项卡的放置位置是用 `JTabbedPane.setTabPlacement` 方法设置的。正如表 12-6 中所示出，当设置选项卡放置位置属性时，这个选项卡窗格无效。这意味着下一次验证选项卡窗格的调用将重新绘制选项卡窗格。因此，在设置选项卡的放置位置后，为选项卡窗格调用了 `validate ()` 方法。

```

...
combo.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        JComboBox cb = (JComboBox) e.getSource ();
        int state = e.getStateChange ();
        if (state == ItemEvent.SELECTED) {
            String s = (String) cb.getSelectedItem ();
            if (s.equals ("TOP"))
                tp.setTabPlacement (JTabbedPane.TOP);
            else if (s.equals ("LEFT"))
                tp.setTabPlacement (JTabbedPane.LEFT);
            else if (s.equals ("RIGHT"))
                tp.setTabPlacement (JTabbedPane.RIGHT);
            else if (s.equals ("BOTTOM"))
                tp.setTabPlacement (JTabbedPane.BOTTOM);
            tp.validate ();
        }
    }
});
...

```

注意 组合框的子项监听器使用了 `JTabbedPane` 前缀来指定选项卡的放置位置，而传递给 `JTabbedPane` 的构造方法的参数则使用了 `SwingConstants` 前缀。重复一下，由于

JTabbedPane 实现 SwingConstants 接口，因此，两个前缀都能够用来指定选项卡的放置位置常数。

例 12-11 完整地列出了图 12-16 中示出的小应用程序的代码。

例 12-11 为 JTabbedPane 的实例设置选项卡放置位置

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JTabbedPane tp =
        new JTabbedPane (SwingConstants.BOTTOM);
    private JComboBox combo = new JComboBox ();

    public Test () {
        Container contentPane = getContentPane ();
        JPanel comboPanel = new JPanel ();
        JPanel panelOne = new JPanel ();
        JPanel panelTwo = new JPanel ();

        tp.add (panelOne, "Panel One");
        tp.addTab ("Panel Two",
            new ImageIcon ("document.gif"),
            panelTwo,
            "tooltip text");

        combo.addItem ("TOP");
        combo.addItem ("LEFT");
        combo.addItem ("RIGHT");
        combo.addItem ("BOTTOM");

        comboPanel.add (new JLabel ("Tab Placement:"));
        comboPanel.add (combo);

        contentPane.add (comboPanel, BorderLayout.NORTH);
        contentPane.add (tp, BorderLayout.CENTER);

        setComboValue ();

        combo.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                JComboBox cb = (JComboBox) e.getSource ();
                int state = e.getStateChange ();

                if (state == ItemEvent.SELECTED) {
                    String s = (String) cb.getSelectedItem ();

                    if (s.equals ("TOP"))
                        tp.setTabPlacement (JTabbedPane.TOP);
                    else if (s.equals ("LEFT"))
                        tp.setTabPlacement (JTabbedPane.LEFT);
                    else if (s.equals ("RIGHT"))
                        tp.setTabPlacement (JTabbedPane.RIGHT);
                    else if (s.equals ("BOTTOM"))
                        tp.setTabPlacement (JTabbedPane.BOTTOM);

                    tp.validate ();
                }
            }
        });
    }
}
```

```
    );  
    |  
private void setComboValue () {  
    int placement = tp.getTabPlacement ();  
    String selectedItem = null;  
    switch (placement) {  
        case JTabbedPane.TOP:  
            selectedItem = "TOP";  
            break;  
        case JTabbedPane.LEFT:  
            selectedItem = "LEFT";  
            break;  
        case JTabbedPane.RIGHT:  
            selectedItem = "RIGHT";  
            break;  
        case JTabbedPane.BOTTOM:  
            selectedItem = "BOTTOM";  
            break;  
    }  
    combo.setSelectedItem (selectedItem);  
    |  
}
```

组件总结 12-4 对 JTabbedPane 类进行了总结。

组件总结 12-1 JTabbedPane

模型: DefaultSingleSelectionModel
UI 代表: javax.swing.plaf.basic.BasicTabbedPaneUI



图 12-17 JTabbedPane 的类图

绘制器： ——
编辑器： ——
激发的事件： PropertyChangeEvent, ChangeEvent
替代： ——
类图： 见图 12-17

JTabbedPane 扩展 JComponent 并实现 SwingConstants、Serializable 和 Accessible 接口。与其他具有一个模型并激发变化事件的 Swing 组件一样，把 ChangeListener 用于响应模型变化，把 ChangeEvent 事件传送给与 JTabbedPane 联系的变化监听器。JTabbedPane 维护一个记录选项卡放置位置的 protected int。JTabbedPane 还维护一个 private boolean 值，这个值记录选项卡窗格是否向工具提示管理器进行了登记。JTabbedPane 还维护一个有包访问范围的向量，以记录一组 Page。Page 是 JTabbedPane 的一个私有类，它封装与选项卡有关的属性。

12.4.2 JTabbedPane 的属性

表 12-6 中列出了 JTabbedPane 的属性。

表 12-6 JTabbedPane 的属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
backgroundAt	Color	SG	I	L&F
boundsAt	Rectangle	G	I	L&F
componentAt	Component	SG	I	null
disabledIconAt	Icon	SG	I	Grayed-out icon
enabledAt	boolean	SG	I	——
foregroundAt	Color	SG	I	L&F
iconAt	Icon	SG	I	——
model	SingleSelection	SG	B	DefaultSingleSele
ction	Model			Model
selectedComponent	Component	SG	S	——
selectedIndex	int	SG	S	——
tabCount	int	G	S	——
tabPlacement	int	CSG	B	TOP
tabRunCount	int	G	S	——
titleAt	String	SG	S	——

① B = 关联的 (激发 PropertyChangeEvent) / C = 受约束的 / I = 索引的 / S = 简单的 / Ch = 激发 ChangeEvent
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ L&F = 与界面样式有关

backgroundAt——为一个用一个索引标识的选项卡指定背景色。如果把一个选项卡的 backgroundAt 属性设置为 null，则这个选项卡的背景色将缺省为这个选项卡所处选项卡窗格的背景色。当这个属性改变时，相应选项卡在选项卡窗格中的区域就被重绘。

boundsAt——一个矩形，代表一个用索引标识的选项卡的边界范围。这个属性是只读的，并且当指定的选项卡不可见时，则 JTabbedPane.getBoundsAt 方法返回 null。

componentAt——代表一个选项卡的组件，这个选项卡是用一个索引标识的。如果 JTabbedPane.getComponentAt () 带一个无效的索引值，则弹出一个 ArrayIndexOutOfBoundsException 异常信息。

disabledIconAt——代表当一个选项卡禁用时显示在这个选项卡中的图标。如果为一个禁用的选项卡设置了属性 disabledIconAt，那么这个选项卡窗格都将重新布置和重绘。

enabledAt——一个 boolean 属性，代表选项卡的启用状态。如果指定的选项卡索引值是无效的，则 `ArrayIndexOutOfBoundsException` 异常。当这个属性改变时，相应选项卡在选项卡窗格中的区域就重绘。

foregroundAt——指定用索引标识的选项卡的前景色。如果一个选项卡的 `foregroundAt` 属性设置为 `null`，则这个选项卡的前景色将缺省为这个选项卡所处选项卡窗格的前景色。当这个属性改变时，相应选项卡在选项卡窗格中的区域就重绘。

iconAt——代表在一个启用的选项卡中显示的图标；这个选项卡是用一个索引值标识的。选项卡图标可以设置为 `null`，在这种情况下，在选项卡中将不会显示任何图标。改变一个选项卡的图标将导致选项卡窗格重新布置和重绘。

model——选项卡窗格的模型是 `SingleSelectionModel` 接口的一个实现。缺省时，选项卡窗格模型是 `DefaultSingleSelectionModel` 接口的一个实现。如果在创建后显式地设置了模型，则将激发一个属性变化事件，并且这个选项卡窗格被重绘。

selectedComponent——选项卡窗格总是有一个已选取的选项卡，而 `selectedComponent` 则代表与这个已选取选项卡相关的组件。可以在程序中用 `JTabbedPane.setSelectedComponent` 方法来设置已选取选项卡。如果传送给这个方法一个空引用，则它将弹出一个 `IllegalArgumentException` 异常信息。

selectedIndex——选项卡既可以用一个索引值选取，也可以用它们的相关组件选取（参见 `selectedComponent` 属性）。`selectedIndex` 是由选项卡窗格的模型维护的，用 `JTabbedPane.setSelectedIndex` 方法设置这个属性将导致模型激发一个变化事件，这个事件将由选项卡窗格的 UI 代表处理，处理方式是重新布置和重绘选项卡窗格。

tabCount——表示包含在一个选项卡窗格中的选项卡数。这个属性是只读的。

tabPlacement——确定选项卡窗格中的选项卡的显式位置。有效值是：

- `JTabbedPane.TOP`
- `JTabbedPane.BOTTOM`
- `JTabbedPane.LEFT`
- `JTabbedPane.RIGHT`

tabRunCount——分别表示具有 `JTabbedPane.TOP/JTabbedPane.BOTTOM` 属性和 `JTabbedPane.LEFT/JTabbedPane.RIGHT` 属性的选项卡的行数或列数。`tabRunCount` 属性是只读的。

titleAt——代表一个选项卡显示的字符串。设置 `titleAt` 属性导致重新布置和重新绘制选项卡窗格。

12.4.3 JTabbedPane 事件

`JTabbedPane` 事件除了在修改它的关联属性时激发属性变化事件外，还在一个选项卡被选取时激发变化事件。由于 `ChangeEvent` 实例维护的唯一状态是事件源，因此，每个 `JTabbedPane` 实例都使用单个 `ChangeEvent` 实例在它的一个选项卡被选取时向它的变化监听器发送信息。

图 12-18 中示出的小应用程序处理由

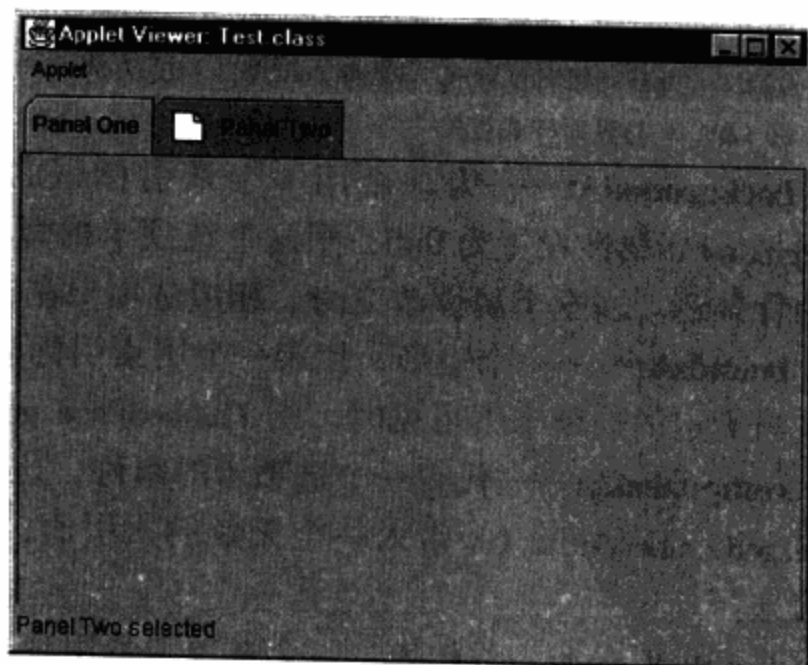


图 12-18 处理 `JTabbedPane` 的变化事件

一个 JTabbedPane 实例激发的变化事件。

例 12-12 列出了图 12-18 中示出的小应用程序的代码。

例 12-12 响应 JTabbedPane 的变化事件

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {

    public Test () {
        Container contentPane = getContentPane ();
        JTabbedPane tp = new JTabbedPane ();
        JPanel panelOne = new JPanel ();
        JPanel panelTwo = new JPanel ();

        tp.add (panelOne, "Panel One");
        tp.addTab ("Panel Two",
                new ImageIcon ("document.gif"),
                panelTwo,
                "tooltip text");

        contentPane.add (tp, BorderLayout. CENTER);

        tp.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                JTabbedPane tabbedPane =
                    (JTabbedPane) e.getSource ();
                int index = tabbedPane.getSelectedIndex ();
                String s = tabbedPane.getTitleAt (index);
                showStatus (s + " selected");
            }
        });
    }
}
```

这个小应用程序在选项卡窗格上添加了一个变化事件，并使用 JTabbedPane 的 `getSelectedIndex` 和 `getTitleAt` 方法在这个小应用程序的状态区中输出选取了哪个选项卡。

12.4.4 JTabbedPane 类总结

类总结 12-4 中列出了 JTabbedPane 类的 `public` 和 `protected` 变量和方法。

类总结 12-4 JTabbedPane

扩展：JComponent

实现：SwingConstants、javax.accessibility.Accessible、java.io.Serializable

1. 构造方法

```
public JTabbedPane ()
public JTabbedPane (int tabPlacement)
```

JTabbedPane 提供了两个构造方法——一个无参数构造方法和一个带一个 `int` 参数值的构造方法。`int` 参数值指定选项卡的放置位置。当使用无参数构造方法时，选项卡的放置位置将缺省地设置为 TOP。

2. 方法

(1) 添加/删除选项卡和组件

```
public Component add (Component)
public Component add (Component, int index)
public void add (Component, Object constraints)
public void add (Component, Object, int index)
public Component add (String, Component)

public void addTab (String title, Icon, Component)
public void addTab (String title, Icon, Component, String title)
public void addTab (String title, Component)

public void insertTab (String title, Icon, Component, String tip, int index)

public void remove (Component)
public void removeAll ()
public void removeTabAt (int index)
```

上面列出的方法是用于添加、插入及删除选项卡和与选项卡相关的组件的。传送给 add 方法的 Object 参数可以是一个代表选项卡标题的字符串，也可以是一个在选项卡中显示的图标。传送给 add 方法的组件参数表示当选取选项卡时将在选项卡下面显示的组件。传送给 add 方法的 int 参数指定选项卡的索引值，也就是选项卡的位置。如果没有为选项卡指定一个标题，将把与选项卡相关的组件的名称用作选项卡的标题。

insertTab 方法带入了两个字符串。第一个字符串指定选项卡的标题，第二个字符串指定与选项卡相关的工具提示文本。

remove 方法从选项卡窗格中删除选项卡及与它们相关的组件。remove (Component) 方法删除组件并通过调用 removeTabAt () 方法删除与被删除组件相关的选项卡。

(2) 属性访问方法

```
public Color getBackgroundAt (int index)
public Rectangle getBoundsAt (int index)
public Component getComponentAt (int index)
public Icon getDisabledIconAt (int index)
public Color getForegroundAt (int index)
public Icon getIconAt (int index)
public SingleSelectionModel getModel ()
public Component getSelectedComponent ()
public int getSelectedIndex ()
public int getTabCount ()
public int getTabPlacement ()
public int getTabRunCount ()
public String getTitleAt (int index)
public String getToolTipText (MouseEvent)

public void setBackgroundAt (int index, Color)
public void setComponentAt (int index, Component)
public void setDisabledIconAt (int index, Icon)
public void setEnabledAt (int index, boolean)
public void setForegroundAt (int index, Color)
public void setIconAt (int index, Icon)
public void setModel (SingleSelectionModel)
public void setSelectedComponent (Component)
public void setSelectedIndex (int index)
public void setTabPlacement (int)
```

```

public void setTitleAt (int index, String)
public boolean isEnabledAt (int)
public int indexOfComponent (Component)
public int indexOfTab (Icon)
public int indexOfTab (String)

```

上面列出的方法是 JTabbedPane 属性的访问方法。传送给大多数方法的 int 值表示一个选项卡的索引值。例如，调用 aTabbedPane.setBackgroundAt (3) 返回索引值为 3 的选项卡的背景色。索引值为 3 的选项卡是选项卡窗格（从索引值为 0 开始）左上角中的第四个选项卡。一个选项卡的索引值可以通过调用 indexOf... 方法得到。

getTabRunCount 方法返回选项卡的列数（如果选项卡放在选项卡窗格的顶部或底部）或选项卡的行数（如果选项卡放在选项卡窗格的左边或右边）。

getToolTipText (MouseEvent) 方法是对 JComponent 的同名方法的重载，返回与一个选项卡相关工具提示文本，这个选项卡是处在鼠标事件发生的位置上的。

有关重载 JComponent.getToolTipText (MouseEvent) 的更多信息参见 4.7.1 节“基于鼠标位置的工具提示”。

(3) 变化监听器

```

public void addChangeListener (ChangeListener)
public void removeChangeListener (ChangeListener)
protected ChangeListener createChangeListener ()
protected void fireStateChanged ()

```

当一个选项卡窗格的选取模型发生了一个变化，即选取了一个选项卡时，则这个选项卡窗格激发一个变化事件，这个变化事件以这个选项卡窗格作为事件源。用上面列出的前两个方法变化把监听器添加到选项卡窗格中或把它们从选项卡窗格中删除。

protected createChangeListener 方法创建一个变化监听器，这个监听器是与选项卡窗格的模型关联的。一旦选项卡窗格的模型发生变化，这个监听器则调用 fireStateChanged 方法。JTabbedPane 的扩展能够通过重载 createChangeListener 方法插入它们自己的监听器，还能够通过重载 fireStateChanged 方法修改激发变化事件的方式。

(4) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public TabbedPaneUI getUI ()
public String getUIClassID ()
public void setUI (TabbedPaneUI)
public void updateUI ()

```

上面方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

12.5 JSplitPane 类

Swing 以 JSplitPane 的形式提供了一个拆分窗格组件，这个组件显示两个组件。在拆分窗格中，组件由一个分隔体隔开，可以拖动这个分隔体以改变组件的大小。包含在一个拆分窗格中组件可以水平放置，也可以垂直放置。

图 12-9 中示出的小应用程序包含了一个 JSplitPane 实例，这个实例包含两个按钮。这个小应用程序还包含了一个控制面板，用以调整一些与这个拆分窗格有关的属性。

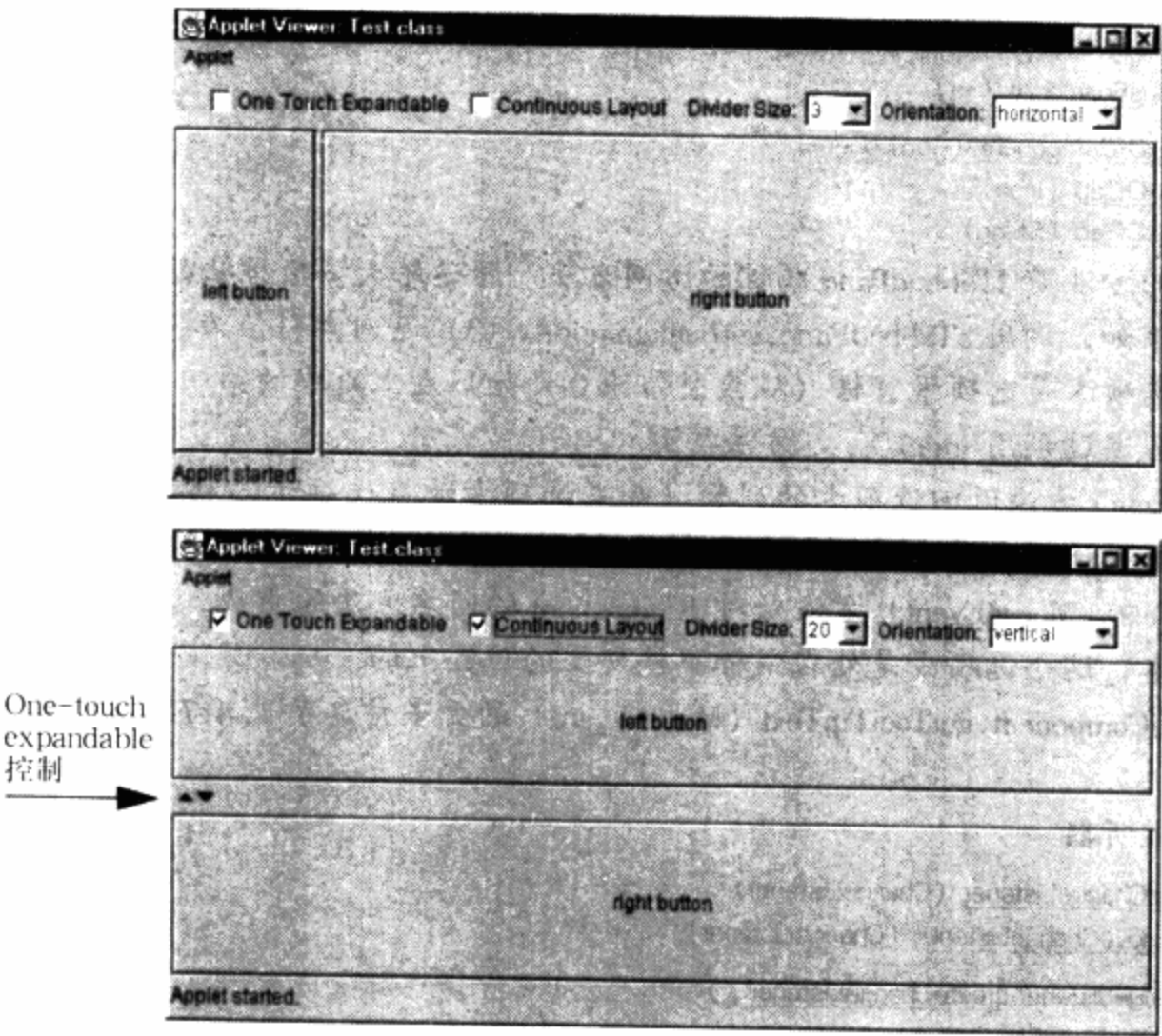


图 12-19 运行中的 JSplitPane

图 12-19 上图示出了这个小应用程序开始时的样子。下图示出了改变属性值后这个小应用程序的样子，这时把 one-touch expandable 和 continuous layout 属性都设置为 true，分隔体的大小设置为 20 个像素点，组件的方向设置为垂直。

Continuous layout 属性控制包含在一个拆分窗格中的组件在分隔体被拖动时是否连续不断地更新。如果这个属性为 true，则组件不断地更新；如果这个属性为 false，则在分隔体拖动到新位置前组件不更新。

one-touch expandable 属性决定是否在分隔体上绘制一个控件。当单击这个控件时，这个控件允许扩展组件或收缩组件。如果 one-touch expandable 属性设置为 true，则绘制这个控件；如果这个属性设置为 false，则这个控件不可见。

这个小应用程序创建拆分窗格和控制面板，并把这个拆分窗格添加到它的内容窗格中的中部。把控制面板添加到这个小应用程序的内容窗格中的上部。

拆分窗格是用 JSplitPane 的无参数构造方法创建的，这个构造方法在拆分窗格中添加了两个按钮。通过 JSplitPane.getTopComponent () 和 JSplitPane.getBottomComponent () 方法从拆分窗格得到了对这两个按钮的引用。虽然这两个按钮最初是水平放置的，但左边的按钮可以用 getTopComponent () 方法获得，右边的按钮则可以用 getBottomComponent () 方法获得。应该指出的是：JSplitPane 还提供了 getRightComponent 方法和 getLeftComponent 方法，但这个小应用程序说明了一个拆分窗格中左组件和右组件能够分别通过 getTopComponent 和 getBottomComponent 方法访问。

在获得了对这个拆分窗格中的两个组件的引用后，打印输出了这两个组件的最小尺寸。这个小应用程序的输出如下所示：

```
left button minimum size: java.awt.Dimension [width = 85, height = 23]
```

right button minimum size: java.awt.Dimension [width = 93, height = 23]

这里输出了最小尺寸，因为 JSplitPane 不允许把其分隔体拖动到这样一个位置上，这个位置得使它的一个组件比这个组件的最小尺寸还小。

```
public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JButton left, right;
        JSplitPane sp = new JSplitPane ();
        ControlPanel cp = new ControlPanel (sp);

        contentPane.add (sp, BorderLayout.CENTER);
        contentPane.add (cp, BorderLayout.NORTH);

        left = (JButton) sp.getTopComponent ();
        right = (JButton) sp.getBottomComponent ();

        System.out.println ("left button minimum size: " +
            left.getMinimumSize ());
        System.out.println ("right button minimum size: " +
            right.getMinimumSize ());
    }
}
```

控制面板是一个 ControlPanel 实例，而 ControlPanel 扩展 JPanel。拆分窗格把 ControlPanel 应用于为控制面板中的每一个控件进行事件处理。

在每个复选框和组合框上都添加了一个子项监听器。“One Touch Expandable”复选框的监听器根据其是否被选取分别以 true 和 false 值调用 JSplitPane 的 setOneTouchExpandable 方法。

“Continuous Layout”复选框的监听器在这个复选框选取时用 true 值调用 JSplitPane.setContinuousLayout () 方法，而当这个复选框未选取时则用 false 调用这个方法。

“Divider Size”复选框的子项监听器调用 JSplitPane.setDividerSize () 方法，根据在这个复选框中选取的字符串传送 integer 值。

“Orientation”复选框的子项监听器则调用 JSplitPane.setOrientation () 方法，根据这个复选框的选取情况分别传送值 JSplitPane.HORIZONTAL_SPLIT 或 JSplitPane.VERTICAL_SPLIT。

```
class ControlPanel extends JPanel {
    private JSplitPane sp;
    ...

    oneTouch.addItemListener (new ItemListener () {
        public void itemStateChanged (ItemEvent e) {
            if (e.getStateChange () == ItemEvent.SELECTED)
                sp.setOneTouchExpandable (true);
            else
                sp.setOneTouchExpandable (false);
        }
    });

    continuous.addItemListener (new ItemListener () {
        public void itemStateChanged (ItemEvent e) {
            if (e.getStateChange () == ItemEvent.SELECTED)
                sp.setContinuousLayout (true);
            else
                sp.setContinuousLayout (false);
        }
    });
}
```

```

dividerSize.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        JComboBox combo = (JComboBox) e.getSource ();
        String s = (String) combo.getSelectedItem ();

        sp.setDividerSize (Integer.parseInt (s));
    }
});
orientation.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        JComboBox combo = (JComboBox) e.getSource ();
        String s = (String) combo.getSelectedItem ();

        if (s.equals ("horizontal"))
            sp.setOrientation (
                JSplitPane.HORIZONTAL_SPLIT);
        else
            sp.setOrientation (JSplitPane.VERTICAL_SPLIT);
    }
});
...
}

```

例 12-13 完整地列出了图 12-19 中示出的小应用程序的代码。

例 12-13 运行中的 JSplitPane

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JButton left, right;
        JSplitPane sp = new JSplitPane ();
        ControlPanel cp = new ControlPanel (sp);

        contentPane.add (sp, BorderLayout.CENTER);
        contentPane.add (cp, BorderLayout.NORTH);

        left = (JButton) sp.getTopComponent ();
        right = (JButton) sp.getBottomComponent ();

        System.out.println ("left button minimum size: " +
            left.getMinimumSize ());
        System.out.println ("right button minimum size: " +
            right.getMinimumSize ());
    }
}

class ControlPanel extends JPanel {
    private JSplitPane sp;

    public ControlPanel (JSplitPane splitPane) {
        sp = splitPane;

        JComboBox dividerSize = new JComboBox ();
        JComboBox orientation = new JComboBox ();
        JCheckBox continuous = new JCheckBox (

```

```

        "Continuous Layout");
JCheckBox oneTouch = new JCheckBox (
        "One Touch Expandable");

Integer initialSize = new Integer (sp.getDividerSize ());
dividerSize.addItem (initialSize.toString ());
dividerSize.addItem ("10");
dividerSize.addItem ("20");
dividerSize.addItem ("30");
dividerSize.addItem ("40");

orientation.addItem ("horizontal");
orientation.addItem ("vertical");

int initialOrientation = sp.getOrientation ();
if (initialOrientation == JSplitPane.HORIZONTAL_SPLIT)
    orientation.setSelectedItem ("horizontal");
else
    orientation.setSelectedItem ("vertical");

boolean initialContinuousLayout = sp.isContinuousLayout ();
if (initialContinuousLayout)
    continuous.setSelected (true);

add (oneTouch);
add (continuous);
add (new JLabel ("Divider Size:"));
add (dividerSize);
add (new JLabel ("Orientation:"));
add (orientation);

oneTouch.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        if (e.getStateChange () == ItemEvent.SELECTED)
            sp.setOneTouchExpandable (true);
        else
            sp.setOneTouchExpandable (false);
    }
});

continuous.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        if (e.getStateChange () == ItemEvent.SELECTED)
            sp.setContinuousLayout (true);
        else
            sp.setContinuousLayout (false);
    }
});

dividerSize.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        JComboBox combo = (JComboBox) e.getSource ();
        String s = (String) combo.getSelectedItem ();

        sp.setDividerSize (Integer.parseInt (s));
    }
});

orientation.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        JComboBox combo = (JComboBox) e.getSource ();

```

```

String s = (String) combo.getSelectedItem();
if (s.equals("horizontal"))
    sp.setOrientation (
        JSplitPane.HORIZONTAL_SPLIT);
else
    sp.setOrientation (JSplitPane.VERTICAL_SPLIT);
}
);

```

组件总结 12-5 对 JSplitPane 组件进行了总结。

组件总结 12-5 JSplitPane

模型：——
 UI 代表： javax.swing.plaf.basic.BasicSplitPaneUI
 绘制器：——
 编辑器：——
 激发的事件： PropertyChangeEvent
 替代：——
 类图：

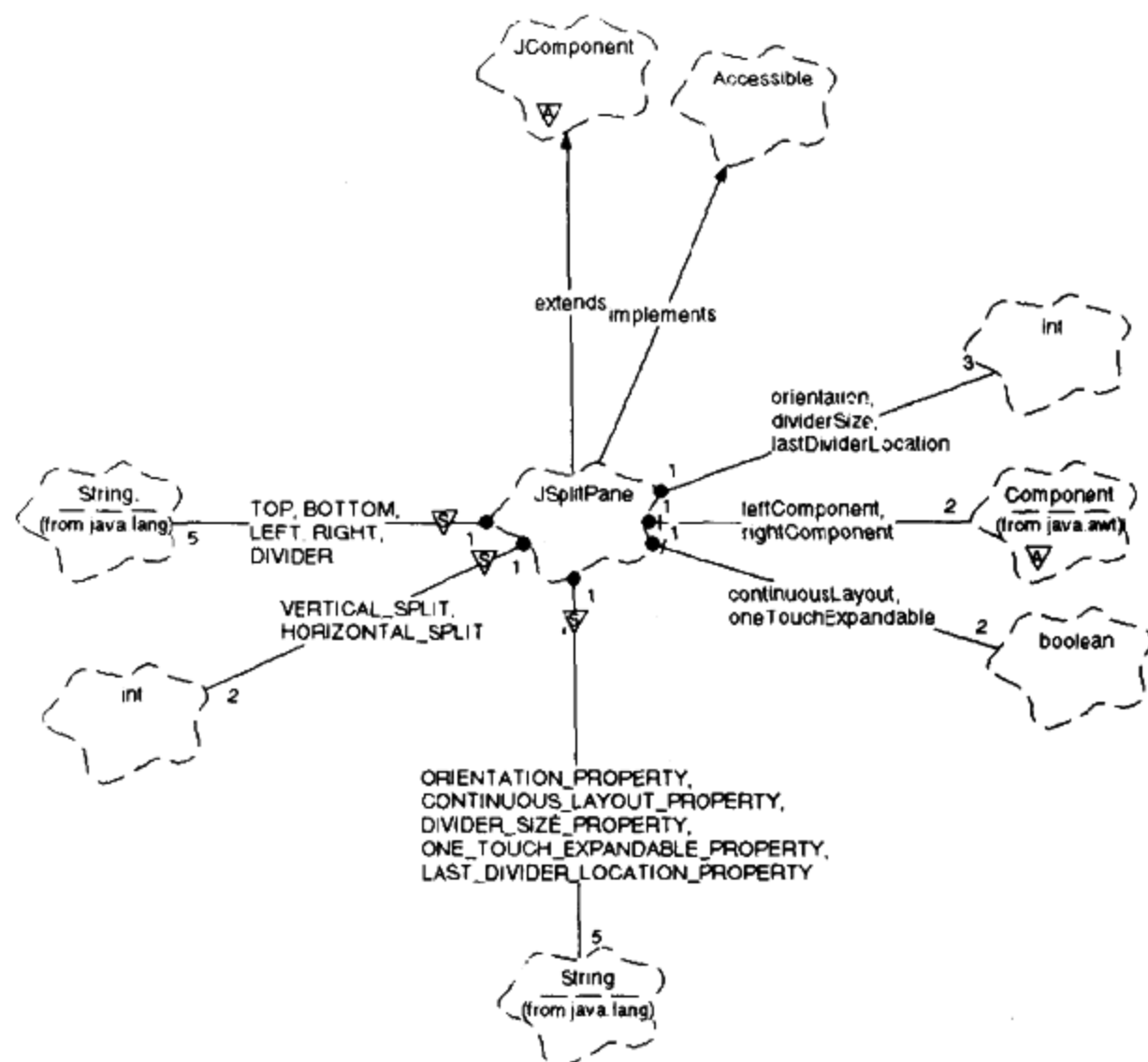


图 12-20 JSplitPane 的类图

JSplitPane 扩展 JComponent 并 Accessible 接口。JSplitPane 定义了很多字符串常数，用于定义

能够添加到一个拆分窗格中的组件类型、方向属性和各种 JSplitPane 类支持的属性。

12.5.1 JSplitPane 属性

表 12-7 中列出了 JSplitPane 维护的属性。

表 12-7 JSplitPane 的属性

属性名	数据类型	访问 ^①	类型 ^②	缺省值 ^③
bottomComponent	Component	CSG	S	null
continuousLayout	boolean	CSG	B	false
dividerLocation (比例)	double	S	S	L&F
dividerLocation (像素位置)	int	SG	S	L&F
dividerSize	int	SG	B	L&F
lastDividerLocation	int	SG	B	L&F
leftComponent	Component	CSG	S	null
minimumDividerLocation	int	G	S	L&F /-1
oneTouchExpandable	boolean	S	B	L&F
orientation	int	CSG	B	水平
rightComponent	Component	CSG	S	null
topComponent	Component	CSG	S	null

- ① C = 可在创建时设置/G = 获取方法/S = 设置方法
- ② B = 关联的(激发 PropertyChangeEvent) /C = 受约束的/ I = 索引的/S = 简单的/Ch = 激发 ChangeEvent
- ③ L&F = 与界面样式有关

bottomComponent——在拆分窗格底部/右边显示的组件。

continuousLayout——如果为 true，两个组件在分隔体移动时连续不断改变大小并重绘。

dividerLocation (比例) ——一个 0.0 到 1.0 之间的值，用于设置分隔条的位置。

dividerLocation (像素位置) ——分隔条以拆分窗格左上角为原点的位置的像素点值。

dividerSize——表示拆分窗格的分隔条的大小（以像素点为单位）。

lastDividerLocation——用像素点表示的分隔条的最后位置，以拆分窗格左上角为原点。

leftComponent——在拆分窗格左边/顶部显示的组件。

minimumDividerLocation——分隔条位置的最小像素值。

orientation——指定拆分窗格中的组件的方向，不指定拆分窗格的分隔条的方向。一个水平拆分窗格水平放置组件，且带有一个垂直分隔体。同样一个垂直拆分窗格垂直放置组件，且带有一个垂直分隔体。

rightComponent——在拆分窗格右边/底部显示的组件。

topComponent——在拆分窗格顶部/左边显示的组件。

12.5.2 JSplitPane 事件

除了在其关联属性被修改时激发属性变化事件外，JSplitPane 类不激发其他事件。

图 12-21 中示出的小应用程序监视一个拆分窗格的分隔体的位置，并在移动分隔体时显示分隔体的位置和包含在这个拆分窗格中的两个按钮的大小。

例 12-14 列出了图 12-21 中示出的小应用程序的代码。

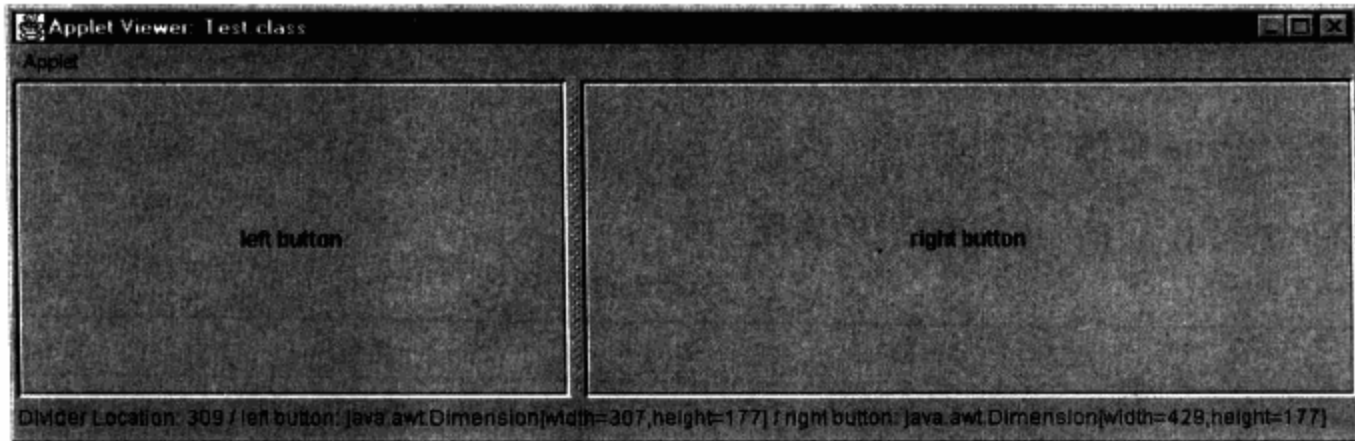


图 12-21 监视分隔体的位置

例 12-14 监视一个拆分窗格的分隔体的位置

```

import java.awt. * ;
import java.awt.event. * ;
import java.beans. * ;
import javax.swing. * ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JSplitPane sp = new JSplitPane ();

        contentPane.add (sp, BorderLayout.CENTER);

        sp.addPropertyChangeListener (
            new PropertyChangeListener () {
                public void propertyChange (PropertyChangeEvent e) {
                    if (e.getPropertyName ().equals (
                        JSplitPane.LAST_DIVIDER_LOCATION_PROPERTY)) {
                        JSplitPane jsp = (JSplitPane) e.getSource ();

                        int dl = jsp.getDividerLocation ();

                        JButton lb = (JButton) jsp.getLeftComponent ();
                        JButton rb = (JButton) jsp.getRightComponent ();

                        showStatus ("Divider Location: " + dl + " / " +
                            lb.getText () + ": " + lb.getSize () + " / " +
                            rb.getText () + ": " + rb.getSize ());
                    }
                }
            }
        );
    }
}

```

这个小应用程序在拆分窗格上添加了一个属性变化监听器。通过比较事件名与 `JSplitPane.LAST_DIVIDER_LOCATION`，这个监听器只响应分隔体最后的位置变化。分隔体的位置是通过调用 `JSplitPane.getDividerLocation()` 方法获得的，通过调用 `JSplitPane` 的 `getLeftComponent()` 和 `getRightComponent()` 方法还获得了对拆分窗格中包含的按钮的引用。

12.5.3 JSplitPane 类总结

类总结 12-5 中列出了 `JSplitPane` 类的 `public` 和 `protected` 变量和方法。

类总结 12-1 JSplitPane

扩展: JComponent

实现: javax.accessibility.Accessible

1. 构造方法:

```
public JSplitPane ()
public JSplitPane (int orientation)
public JSplitPane (int orientation, boolean continuousLayout)
public JSplitPane (int orientation, boolean continuousLayout, Component top/left, Component bottom/right)
public JSplitPane (int orientation, Component top/left, Component bottom/right)
```

无参数构造方法在拆分窗格中添加了两个按钮。提供这个构造方法主要是为 JavaBean 的开发人员提供演示某些有趣性能的工具。这个无参数构造方法把连续布局属性设置为 false 并把组件的方向设置为水平,正如表 12-7 中列出的那样。

传送给上面列出的构造方法的两个组件分别代表顶部/左边组件和底部/右边组件。

integer 值指定拆分窗格的方向: JSplitPane.HORIZONTAL_SPLIT 和 JSplitPane.VERTICAL_SPLIT 是有效值。如果为方向指定的值不是 HORIZONTAL_SPLIT 或 VERTICAL_SPLIT,那么将弹出一个 IllegalArgumentException 异常信息。

boolean 值为拆分窗格指定连续布局属性。如果连续布局为 true,则拆分窗格中的两个将在分隔体移动的过程中改变大小。如果连续布局为 false,那么拆分窗格中的两个将在分隔体在分隔体到达一个新位置后才改变大小。

2. 方法

(1) 添加组件/绘制子组件

```
protected void addImpl (Component, Object constraints, int index)
protected void paintChildren (Graphics)
```

JTabbedPane 重载上面列出的 protected 组件,以确保以正确的顺序把组件添加到拆分窗格中并确保当拆分窗格完成绘制它的子组件时 UI 代表能够得到通知。上面列出的方法很少被 JSplitPane 的扩展重载。

(2) 删除组件

```
public void remove (int index)
public void remove (Component)
public void removeAll ()
public void resetToPreferredSize ()
```

这些删除方法是重载 java.awt.Container 中的方法得到的,以便 JSplitPane 能够正确地设置它的 leftComponent 和 rightComponent 实例变量。resetToPreferredSize 方法强制拆分窗格根据其所包含的组件的首选大小重新布局。

(3) 属性访问方法

```
public Component getBottomComponent ()
public int getDividerLocation ()
public int getDividerSize ()
public int getLastDividerLocation ()
public Component getLeftComponent ()
public int getMaximumDividerLocation ()
public int getMinimumDividerLocation ()
public int getOrientation ()
public Component getRightComponent ()
```

```
public Component getTopComponent ()
public void setBottomComponent (Component)
public void setContinuousLayout (boolean)
public void setDividerLocation (double)
public void setDividerLocation (int)
public void setDividerSize (int)
public void setLastDividerLocation (int)
public void setLeftComponent (Component)
public void setOneTouchExpandable (boolean)
public void setOrientation (int)
public void setRightComponent (Component)
public void setTopComponent (Component)

public boolean isContinuousLayout ()
public boolean isOneTouchExpandable ()
```

上面列出的是 JSplitPane 属性的访问方法。

(4) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public void setUI (SpGtPaneui)
public void updateUI ()
public SplitPaneUI getUI ()
public String getUIClass ()
```

上面方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

12.5.4 AWT 兼容

AWT 没有提供与 JSplitPane 类似的组件。

12.6 本章回顾

Swing 提供了丰富的基本容器，从简单的 JPanel 组件到较复杂的 JLayeredPane、JTabbedPane 和 JSplitPane 组件。本章介绍了基本的 Swing 容器；但是，Swing 还提供了其他一些容器，如 JViewport 和 JScrollPane 类，它们将在第 13 章“滚动”中介绍。

第 13 章 滚 动

本章介绍 Swing 的滚动体系结构，这个体系结构包括两个轻量 Swing 容器、一个 Scrollable 接口、和一个 JScrollBar 类。其中，这两个轻量 Swing 容器是 JViewport 和 JScrollPane，设计这个 Scrollable 接口来支持有特殊滚动需求的组件。

JViewport 实例很少被实例化，也很少被直接使用，然而，本章仍将用相当大的篇幅来介绍 JViewport 类，因为它是 Swing 滚动体系结构的基础组件。

设计 JScrollPane 组件来替代 AWT 的重量组件 ScrollPane。JScrollPane 在 AWT 的 ScrollPane 基础上做了许多改进，包括具有配置行头部和列头部的能力和具有指定在滚动窗格角部组件的能力。

Scrollable 接口是为表格、树、文本组件和列表等包含数据行或数据列的可滚动组件而设计的。

JScrollBar 组件是一个用来实现手动滚动的滚动条。虽然 Swing 的 JScrollPane 组件在大多数滚动情况是足够用了，但是，有时为了性能或资源的考虑，还必须实现手动滚动。在这种情况下，可以用 JScrollBar 组件来滚动容器的内容。

13.1 JViewport

JViewport 类是 Swing 滚动体系结构的基础。就如它的名字所指出的那样，JViewport 的实例提供一个视口，视图的特定区域可以显示在这个视口中。可以操纵由视口显示的视图的位置，以便在不同的时刻，使不同的视图区域出现在这个视口中。

图 13-1 所示的小应用程序包含 JViewport 的一个实例，它的视图是 JLabel 的一个实例。该标签配备了一幅比显示图像的视口还要大的图像。这个小应用程序提供了四个用于调整视口中视图位置的按钮。

图 13-1 中的左图显示这个小应用程序开始时的样子。右图显示在反复激活 up 和 left 按钮后，这个小应用程序的样子。应该强调的是，反复激活按钮不是滚动视口中视图最理想的方式，但是它说明了一个视口是怎样调整其视图位置的。

这个小应用程序创建 JViewport 的一个实例和 JLabel 的一个实例。通过调用 JViewport.setVi

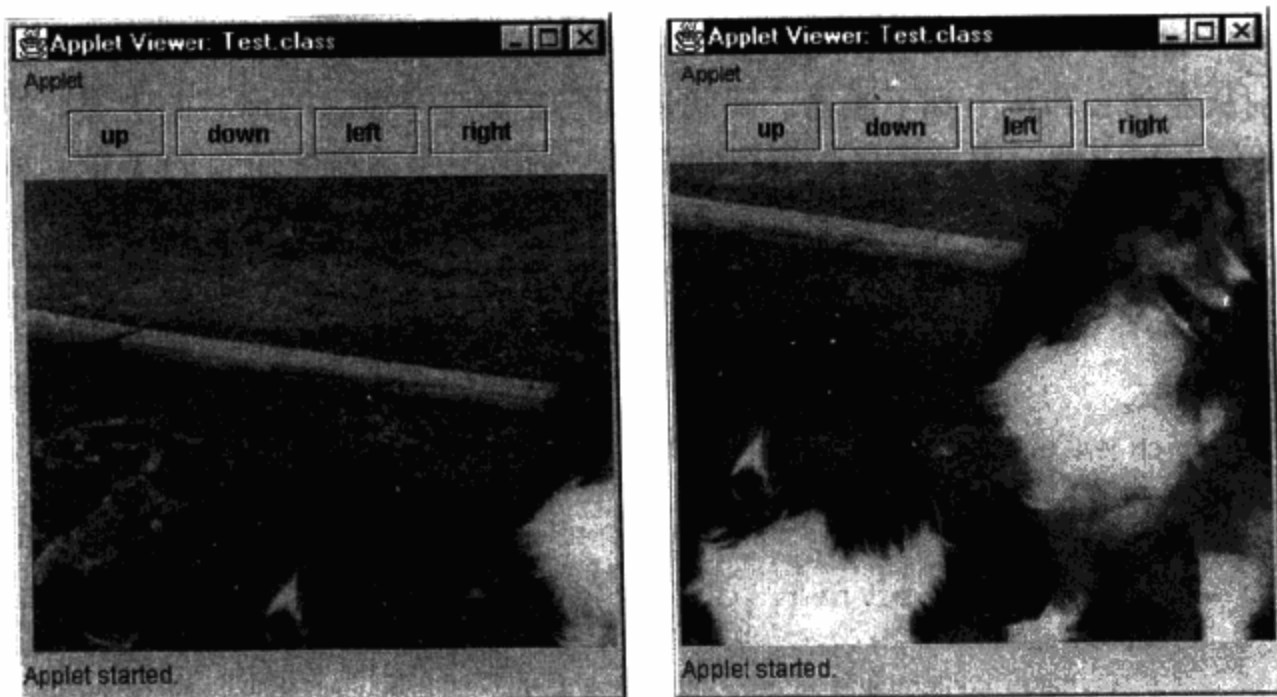


图 13-1 使用 JViewport 来滚动一幅图像

ew() 来指定这个标签作为视口的视图。这个小应用程序还创建了一个 ControlPanel 实例（它是 JPanel 的一个扩展），这个实例包含了四个用于滚动标签的按钮。

```
public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JViewport viewport = new JViewport ();
        JPanel view = new JPanel ();

        view.add (new JLabel (
            new ImageIcon ("anjinAndMariko.gif")));

        viewport.setView (view);

        contentPane.add (new ControlPanel (viewport),
            BorderLayout.NORTH);
        contentPane.add (viewport, BorderLayout.CENTER);
    }
    ...
}
```

ControlPanel 类创建四个按钮并把单个 ActionListener 实例添加到每个按钮中。这个监听器调用一个 private scroll 方法，该方法以与激活的按钮相关联的动作命令字符串为参数。scroll 方法调用 JViewport.getViewPosition() 来获得视图的当前位置，接着调用 JViewport.setViewPosition() 来调整视图的位置。JViewport.setViewPosition 方法带一个点，这个点代表视口中显示的视图左上角的坐标。

```
class ControlPanel extends JPanel {
    private JViewport viewport;
    ...

    private void scroll (String actionCmd) {
        Point vp = viewport.getViewPosition ();
        if (actionCmd.equals ("up")) vp.y += 5;
        else if (actionCmd.equals ("down")) vp.y -= 5;
        else if (actionCmd.equals ("left")) vp.x += 5;
        else if (actionCmd.equals ("right")) vp.x -= 5;

        viewport.setViewPosition (vp);
    }
}
```

Swing（和 AWT）组件的坐标系统定义原点（0，0）在这个组件的左上角，X 值和 Y 值分别从左到右和从上到下递增。因此，要把在视口中显示的标签向上移，则增加当前视图位置的 Y 值。相应地，要把视口的视图向下移，则减少当前视图位置的 Y 值。要把视图向左移，就增加视图位置的 X 值，要把视图向右移，就减少视图位置的 X 值。

例 13-1 列出了图 13-1 所示小应用程序的完整代码。

例 13-1 用 JViewport 的一个实例来滚动一幅图像

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JViewport viewport = new JViewport ();
```

```

JPanel view = new JPanel ();
view.add (new JLabel (
    new ImageIcon ("anjinAndMariko.gif")));
viewport.setView (view);
contentPane.add (new ControlPanel (viewport),
    BorderLayout.NORTH);
contentPane.add (viewport, BorderLayout.CENTER);
}
}
class ControlPanel extends JPanel {
    private JViewport viewport;
    private JButton [] buttons = {
        new JButton ("up"), new JButton ("down"),
        new JButton ("left"), new JButton ("right")
    };
    public ControlPanel (JViewport vp) {
        viewport = vp;
        for (int i=0; i < buttons.length; ++i) {
            add (buttons [i]);
            buttons [i] .addActionListener (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    scroll (e.getActionCommand ());
                }
            });
        }
    }
    private void scroll (String actionCmd) {
        Point vp = viewport.getViewPosition ();
        if (actionCmd.equals ("up")) vp.y += 5;
        else if (actionCmd.equals ("down")) vp.y -= 5;
        else if (actionCmd.equals ("left")) vp.x += 5;
        else if (actionCmd.equals ("right")) vp.x -= 5;
        viewport.setViewPosition (vp);
    }
}
}

```

Swing 提示

视口与它们的视图

Swing 视口显示组件的一部分，这个组件称作视口的视图。JViewport 类维护视图位置属性，这个属性代表在视口左上角显示的视图坐标。还为视图位置属性提供了设置和获取的方法。

Swing (和 AWT) 组件的坐标系统定义坐标 (0, 0) 在这个组件的左上角，X 和 Y 值分别从左到右递增和从上到下递增。因此，用下面的原则来移动一个视口的视图：

- 把视图向上移动：增加视图的 Y 坐标。
- 把视图向下移动：减少视图的 Y 坐标。
- 把视图向左移动：增加视图的 X 坐标。
- 把视图向右移动：减少视图的 X 坐标。

13.1.1 拖动视口中的视图

可以用 JViewport 的 `setViewPosition` 方法和 `getViewPosition` 方法来简单地实现一个监听器，把这个监听器添加到视口中，以便拖动视口中的视图。图 13-2 所示的小应用程序实现了这样一个监听器。



图 13-2 在一个视口中拖动一副图像

图 13-2 中的左图显示这个小应用程序开始时的样子。右图显示拖动视口的视图后小应用程序的样子。

这个小应用程序实现一个监听器，这个监听器扩展 `java.awt.event.MouseAdapter` 类并实现 `java.awt.event.MouseMotionListener` 接口。这个监听器的 `mousePressed` 方法记录鼠标按下事件的位置，而 `mouseDragged` 方法计算鼠标拖动事件的位置与前一个鼠标事件位置之间的偏移。从当前视图位置中提取这个偏移，并用这个偏移来重新设置视口的视图位置。

```
class ViewportDragListener extends MouseAdapter
    implements MouseMotionListener {
    private JViewport viewport;
    private Point last = new Point (), scrollTo = new Point ();

    public ViewportDragListener (JViewport viewport) {
        this.viewport = viewport;
    }

    public void mousePressed (MouseEvent e) {
        last.x = e.getPoint ().x;
        last.y = e.getPoint ().y;
    }

    public void mouseMoved (MouseEvent e) {
        // must be implemented for ViewportDragListener
        // to implement the MouseMotionListener interface
    }

    public void mouseDragged (MouseEvent e) {
        Point drag = e.getPoint ();
        Point viewPos = viewport.getViewPosition ();
        Point offset = new Point (drag.x-last.x, drag.y-last.y);
```

```

last.x = drag.x;
last.y = drag.y;

if (viewport.contains (drag)) {
    scrollTo.x = viewPos.x - offset.x;
    scrollTo.y = viewPos.y - offset.y;
    viewport.setViewPosition (scrollTo);
}
}

```

例 13-2 列出了图 13-2 所示的小应用程序的代码。

例 13-2 拖动视口中的视图

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JLabel label = new JLabel (new ImageIcon ("pic.gif"));
        JViewport vp = new JViewport ();
        ViewportDragListener listener =
            new ViewportDragListener (vp);

        vp.setView (label);
        vp.addMouseListener (listener);
        vp.addMouseMotionListener (listener);
        contentPane.add (vp, BorderLayout.CENTER);
    }
}

class ViewportDragListener extends MouseAdapter
    implements MouseMotionListener {
    private JViewport viewport;
    private Point last = new Point (), scrollTo = new Point ();

    public ViewportDragListener (JViewport viewport) {
        this.viewport = viewport;
    }

    public void mousePressed (MouseEvent e) {
        last.x = e.getPoint ().x;
        last.y = e.getPoint ().y;
    }

    public void mouseMoved (MouseEvent e) {
        // must be implemented for ViewportDragListener
        // to implement the MouseMotionListener interface
    }

    public void mouseDragged (MouseEvent e) {
        Point drag = e.getPoint ();
        Point viewPos = viewport.getViewPosition ();
        Point offset = new Point (drag.x-last.x, drag.y-last.y);
        last.x = drag.x;
        last.y = drag.y;

        if (viewport.contains (drag)) {
            scrollTo.x = viewPos.x - offset.x;

```

```
scrollTo.y = viewPos.y - offset.y;
viewport.setViewPosition (scrollTo);
```

13.1.2 使用 scrollRectToVisible 方法

JViewport 类重载 JComponent 类的 scrollRectToVisible 方法来滚动视图，使传送给这个方法的矩形是可见的。然而，一般很少直接调用 JViewport.scrollRectToVisible 方法，而是为视口的视图调用 scrollRectToVisible 方法。

图 13-3 所示的小应用程序创建一个有 18 个按钮的数组，这些按钮包含在一个面板中，这个面板被指定为这个小应用程序内容窗格的居中组件。从组合框中选取一个按钮，将导致所选的这个按钮滚动到视图中。图 13-3 中的左图显示这个小应用程序开始时的样子，右图显示从组合框中选取了 15 号按钮后这个小应用程序的样子。

GridLayout 按三列来布局按钮，并设置程序的大小以便在任意给定时刻只有两列是可见的。

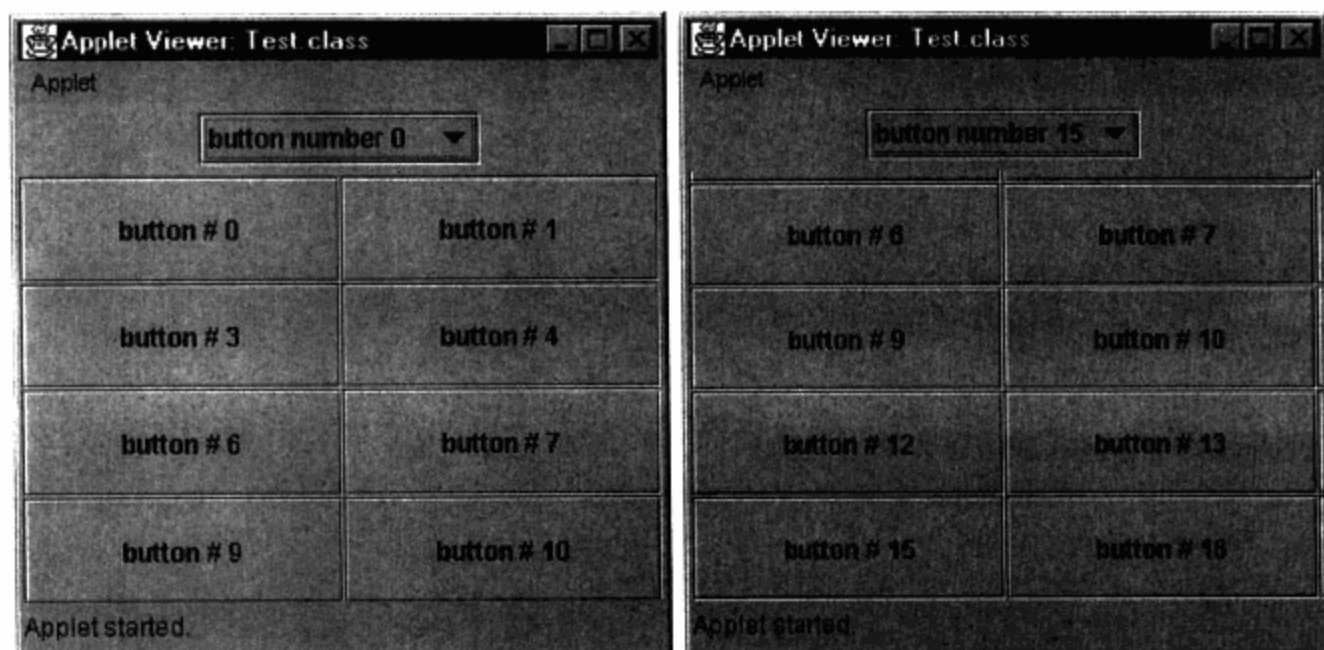


图 13-3 为视口的视图调用 scrollRectToVisible 方法

这个小应用程序创建 JViewport 的一个实例和视口的视图，这个视图是 JPanel 的一个实例。视图的布局管理器被设置为 GridLayout 的一个实例，构造这个实例，以便它按三列来布局这些按钮。把这些按钮初始化并添加到视图中，通过调用 JViewport.setView() 把这个视图指定为视口的视图。这个小应用程序还创建一个包含组合框的 ControlPanel 实例 (JPanel 的一个扩展)。最后，把控制面板和视口添加到这个小应用程序的内容窗格中。

```
public class Test extends JApplet {
    private JButton buttons [] = {
        ...
    }
    public void init () {
        Container contentPane = getContentPane ();
        JViewport viewport = new JViewport ();
        JPanel view = new JPanel ();
        JPanel controlPanel = new ControlPanel (buttons);
```

```

view.setLayout (new GridLayout (0, 3));
for (int i=0; i < buttons.length; ++i) {
    buttons [i] .setText ("button # " + i);
    buttons [i] .setPreferredSize (new Dimension (150, 50));
    view.add (buttons [i]);
}

viewport.setView (view);

contentPane.add (controlPanel, BorderLayout.NORTH);
contentPane.add (viewport, BorderLayout.CENTER);
}
}

```

控制面板把一个项监听器添加到组合框中，这个组合框根据在组合框中所做的选取来获得对一个按钮的引用。

然后，调用这个按钮的 `scrollRectToVisible` 方法，为要滚动到视图中的矩形指定这个按钮的左上角和这个按钮的宽度和高度。

```

class ControlPanel extends JPanel {
    private JComboBox combo = new JComboBox ();

    public ControlPanel (final JButton [] buttons) {
        add (combo);

        for (int i=0; i < buttons.length; ++i) {
            combo.addItem ("button number " + i);
        }

        combo.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                int index = combo.getSelectedIndex ();
                JButton button = buttons [index];
                Dimension size = button.getSize ();

                buttons [index] .scrollRectToVisible (
                    new Rectangle (0, 0, size.width, size.height));
            }
        });
    }
}

```

当为一个按钮调用 `scrollRectToVisible` 时，`JComponent.scrollRectToVisible` 被调用，因为 `JButton` 类不重载 `scrollRectToVisible`。`scrollRectToVisible` 的 `JComponent` 实现对包含在一个组件的容器层次结构中的容器进行遍历，为每个容器调用 `scrollRectToVisible`。如果这个容器没有重载 `scrollRectToVisible`，则左上角的坐标被转换成这个容器的坐标系统，然后把这个请求传送给下一个容器。

重载 `JViewport.scrollRectToVisible` 以滚动传送给视图的矩形。结果，当为图 13-3 示出的小应用程序中的一个按钮调用 `scrollRectToVisible` 时，就为这个按钮的容器层次结构中的每个容器调用 `scrollRectToVisible`，直到找到 `JViewport` 实例，然后视口把这个按钮滚动到视图中。

例 13-3 列出了图 13-3 所示小应用程序的完整代码。

例 13-3 使用 `scrollRectToVisible()`

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

```

```

public class Test extends JApplet {
    private JButton buttons [] = {
        new JButton (), new JButton (), new JButton (),
        new JButton (), new JButton (), new JButton (),
        new JButton (), new JButton (), new JButton (),
        new JButton (), new JButton (), new JButton (),
        new JButton (), new JButton (), new JButton (),
        new JButton (), new JButton (), new JButton (),
    };

    public void init () {
        Container contentPane = getContentPane ();
        JViewport viewport = new JViewport ();
        JPanel view = new JPanel ();
        JPanel controlPanel = new ControlPanel (buttons);

        view.setLayout (new GridLayout (0, 3));

        for (int i=0; i < buttons.length; ++i) {
            buttons [i] .setText ("button # " + i);
            buttons [i] .setPreferredSize (new Dimension (150, 50));
            view.add (buttons [i]);
        }

        viewport.setView (view);
        contentPane.add (controlPanel, BorderLayout.NORTH);
        contentPane.add (viewport, BorderLayout.CENTER);
    }
}

class ControlPanel extends JPanel {
    private JComboBox combo = new JComboBox ();

    public ControlPanel (final JButton [] buttons) {
        add (combo);

        for (int i=0; i < buttons.length; ++i) {
            combo.addItem ("button number " + i);
        }

        combo.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                int index = combo.getSelectedIndex ();
                JButton button = buttons [index];
                Dimension size = button.getSize ();

                buttons [index] .scrollRectToVisible (
                    new Rectangle (0, 0, size.width, size.height));
            }
        });
    }
}

```

Swing 提示

Swing 组件和 scrollRectToVisible 方法

scrollRectToVisible 方法在 JComponent 类中被定义，因此，任何 Swing 轻量组件都可以调用它。当为一个包含在视口中或文本域中的轻量 Swing 组件调用 scrollRectToVisible 时，把与这个组件有关的指定矩形滚动到视图中。

传送给 `scrollRectToVisible()` 的矩形与组件有关，而不与组件的容器有关，这点是很重要的。例如，要把组件的整个边缘都滚动到视图中，应该像下面这样调用 `scrollRectToVisible()`：

```
Dimension size = aComponent.getSize ();
aComponent.scrollRectToVisible (0, 0, size.width, size.height);
```

组件总结 13-1 总结了 `JViewport` 组件。

组件总结 13-1 JViewport

模型：——
 UI 代表：——
 绘制器：——
 编辑器：——
 激发的事件：ChangeEvents
 替换为：——
 类图：见图 13-4

`JViewport` 类扩展 `JComponent` 并实现 `Accessible` 接口。与大多数轻量 Swing 容器相似，`JViewport` 没有模型和 UI 代表。

`JViewport` 的实例维护对代表视图位置点的一个 `protected` 引用。这里，视图位置指最后一次绘制视口后视图的位置。

`JViewport` 维护对 `java.awt.Image` 实例的 `protected` 引用，这个 `java.awt.Image` 实例被用作视图的屏外缓冲。另外，`JViewport` 还维护 `boolean` 引用，它跟踪是否使用了屏外缓存来绘制视图。

`protected isVisibleSizeSet` 成员跟踪是否设置了这个视图的大小。如果已经设置了这个视图的大小，则 `JViewport.getViewSize` 返回视图的实际大小，否则，它返回这个视图的首选大小。如果没有设置这个视口的视图，则 `JViewport.getViewSize` 返回值是 (0, 0)。

用 `scrollUnderway` `boolean` 成员来确保 `JViewport` 的实例能自动滚动（参见 4.6 节“自动滚动”）。

当视口的视图改变大小时，对 `ChangeListener` 的一个实例

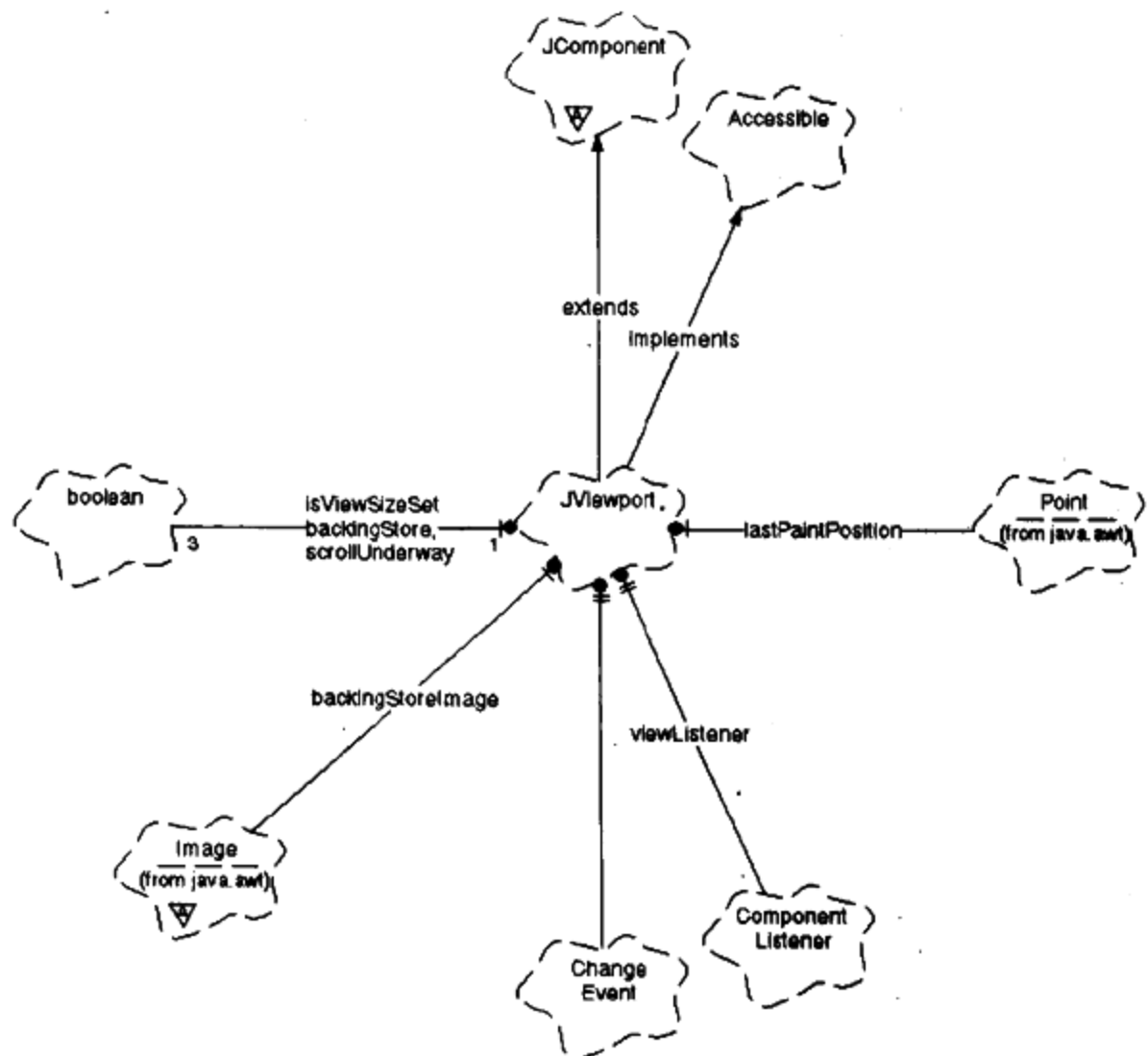


图 13-4 JViewport 类图

的 private 引用被用于激发变化事件。单个 ChangeEvent 实例用于向监听器广播变化事件。

13.1.3 JViewport 属性

表 13-1 列出了由 JViewport 类维护的属性。

表 13-1 JViewport 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
backingStoreEnabled	boolean	S	SG	false
extentSize	Dimension	Ch	SG	——
view	Component	S	SG	——
viewPosition	Point	Ch	SG	——
viewRect	Rectangle	——	G	——
viewSize	Dimension	Ch	SG	——

① B = 关联的 (激发 PropertyChangeEvent) / C = 受约束的 / I = 索引的 / S = 简单的 / Ch = 激发 ChangeEvent
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ I&F = 与界面样式有关

backingStoreEnabled——如果为 true，则当视图位置（相对于视口）在一个方向上有小的增加时，就使用屏外缓存来绘制视图。

extentSize——代表视图可见部分的一个 Dimension 实例。

view——在视口中显示的组件。

viewPosition——一个点，代表视口左上角的视图坐标。

viewRect——Rectangle 的一个实例，代表视图可见部分的大小和位置。矩形的宽度和高度和 extentSize 的大小相同。

viewSize——Dimension 的一个实例，代表视图的大小。如果没有显式地设置视图的大小，则 viewSize 的缺省值是视图的首选大小。如果没有视图，则视图的大小是 (0, 0)。

13.1.4 JViewport 事件

当修改视口的 viewPosition、viewSize 和 extentSize 属性时，JViewport 的实例将激发变化事件。图 13-5 所示的小应用程序除了显示视图的可见部分的大小、视图大小和视图位置外，其他内容几乎与图 13-1 所示的小应用程序的内容完全相同。

这个小应用程序指定 StatusPanel 的一个实例作为其内容窗格下边的组件。StatusPanel 类是 JPanel 的一个扩展，它包含三个标签，它们由 BoxLayout 的一个实例沿着 Y 轴布局。由于这些标签沿 X 轴的排列设置为 0.5，所以这些标签是居中的，有关用 BoxLayout 的实例布局组件的详细内容，请参见 6.5.1 节“BoxLayout 类”。

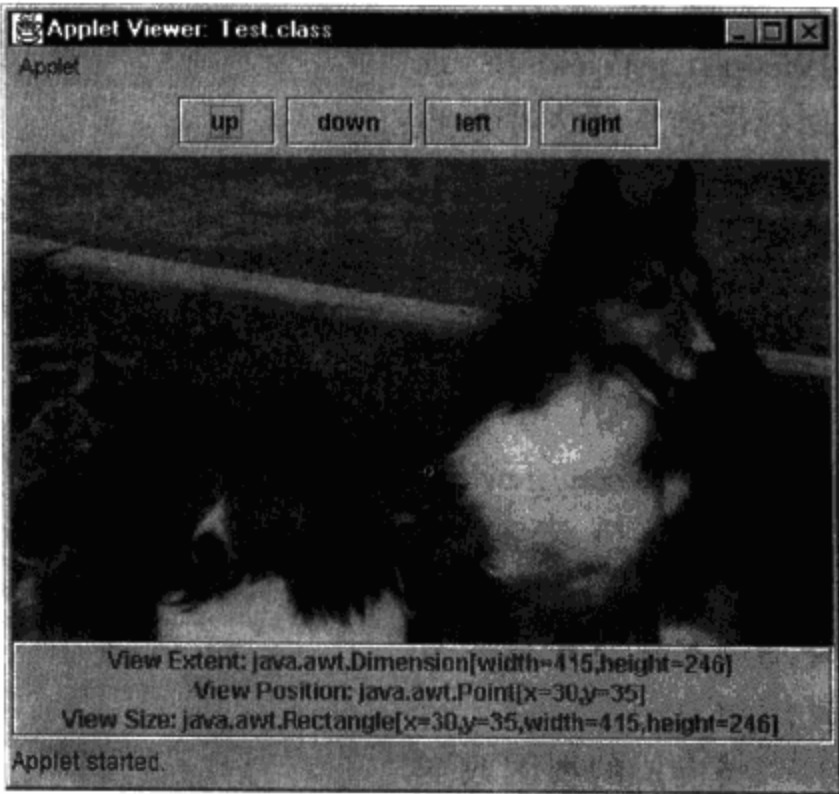


图 13-5 响应 JViewport 变化事件

这个小应用程序把 `ChangeListener` 的一个实例添加到视口中,以便更新 `StatusPanel` 中的这些标签;因为这些标签在修改它们所代表的属性时,可能会改变大小,所以这个小应用程序调用 `revalidate()`,以确保这些标签被布局 and 重新绘制。有关 `revalidate` 方法的更多信息,请参见 4.3.5 节“`validate`、`invalidate` 和 `revalidate` 方法”。

```
class StatusPanel extends JPanel {
    private JLabel extentLabel = new JLabel (),
        viewPositionLabel = new JLabel (),
        viewSizeLabel = new JLabel ();
    private JViewport viewport;

    public StatusPanel (JViewport vp) {
        viewport = vp;

        setBorder (BorderFactory.createEtchedBorder ());
        setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));

        extentLabel.setAlignmentX (0.5f);
        viewPositionLabel.setAlignmentX (0.5f);
        viewSizeLabel.setAlignmentX (0.5f);

        add (extentLabel);
        add (viewPositionLabel);
        add (viewSizeLabel);

        viewport.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                Dimension extent = viewport.getExtentSize ();
                Point viewPosition = viewport.getViewPosition ();
                Rectangle viewSize = viewport.getViewRect ();

                extentLabel.setText ("View Extent: " +
                                    extent.toString ());
                viewPositionLabel.setText ("View Position: " +
                                           viewPosition.toString ());
                viewSizeLabel.setText ("View Size: " +
                                       viewSize.toString ());

                revalidate ();
            }
        });
    }
}
```

例 13-4 列出了图 13-5 所示小应用程序的完整代码。

例 13-4 对 `JViewport` 变化事件的响应

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JViewport viewport = new JViewport ();
        JPanel view = new JPanel ();

        view.add (new JLabel (
```

```

        new ImageIcon ("anjinAndMariko.gif"));
    viewport.setView (view);
    contentPane.add (new ControlPanel (viewport),
        BorderLayout.NORTH);
    contentPane.add (viewport, BorderLayout.CENTER);
    contentPane.add (new StatusPanel (viewport),
        BorderLayout.SOUTH);
}
}
class StatusPanel extends JPanel {
    private JLabel extentLabel = new JLabel (),
        viewPositionLabel = new JLabel (),
        viewSizeLabel = new JLabel ();
    private JViewport viewport;
    public StatusPanel (JViewport vp) {
        viewport = vp;

        setBorder (BorderFactory.createEtchedBorder ());
        setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));

        extentLabel.setAlignmentX (0.5f);
        viewPositionLabel.setAlignmentX (0.5f);
        viewSizeLabel.setAlignmentX (0.5f);

        add (extentLabel);
        add (viewPositionLabel);
        add (viewSizeLabel);

        viewport.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                Dimension extent = viewport.getExtentSize ();
                Point viewPosition = viewport.getViewPosition ();
                Rectangle viewSize = viewport.getViewRect ();

                extentLabel.setText ("View Extent: " +
                    extent.toString ());
                viewPositionLabel.setText ("View Position: " +
                    viewPosition.toString ());
                viewSizeLabel.setText ("View Size: " +
                    viewSize.toString ());

                invalidate ();
                validate ();
            }
        });
    }
}
class ControlPanel extends JPanel {
    private JViewport viewport;
    private JButton [] buttons = {
        new JButton ("up"), new JButton ("down"),
        new JButton ("left"), new JButton ("right"),
    };
    public ControlPanel (JViewport vp) {
        viewport = vp;

        for (int i = 0; i < buttons.length; ++ i) {

```

```

        add (buttons [i]);
        buttons [i] .addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                scroll (e.getActionCommand ());
            }
        });
    }

    private void scroll (String actionCmd) {
        Point vp = viewport.getViewPosition ();
        if (actionCmd.equals ("up")) vp.y += 5;
        else if (actionCmd.equals ("down")) vp.y -= 5;
        else if (actionCmd.equals ("left")) vp.x += 5;
        else if (actionCmd.equals ("right")) vp.x -= 5;
        viewport.setViewPosition (vp);
    }
}

```

13.1.5 JViewport 类总结

类总结 13-1 列出了 JViewport 的 public 和 protected 变量和方法。

类总结 13-1 JViewport

扩展: JComponent

实现: javax.accessibility.Accessible

1. 构造方法

public JViewport ()

JViewport 只提供一个无参数的构造方法, 即只有在视口被构造后, 才可以指定一个视口的视图。

2. 方法

(1) 变化事件

public void addChangeListener (ChangeListener)

protected void fireStateChanged ()

public void removeChangeListener (ChangeListener)

当修改 JViewport 的关联属性时, JViewport 将激发变化事件。用 addChangeListener 方法把变化监听器添加到 JViewport 的实例中, 用 removeChangeListener 方法删除变化监听器。用 protected fireStateChanged 方法把变化事件发送给登记了的变化监听器。JViewport 的扩展可以重载 fireStateChanged 方法来修改变化监听器接收属性变化通知的方式, 但是实际中很少重载这个方法。

(2) protected 实用方法

protected void addImpl (Component, Object, int)

protected boolean computeBlit (int dx, int dy, Point blitFrom,
Point blitTo, Dimension blitSize, Rectangle blitPaint)

protected LayoutManager createLayoutManager ()

protected JViewport.ViewListener createViewListener ()

JViewport 是一个总是包含单个组件 (称作视图) 的轻量 Swing 组件。因此, JViewport 类重载 addImpl 方法 (来自 java.awt.Container 类), addImpl 方法删除当前的视图并用指定的组件来进行替代。因为 JViewport 重载 addImpl (), 所以, 可以调用 JViewport.setView 或 JViewport.add (a-

Component) 来指定一个视口的视图。

`computeBlit()` 计算把视图的图像从屏外缓存拷贝到屏上代表中所需要的参数。JViewport 的扩展应该很少 (如果有的话) 有原因要重载或调用这个方法。

`createLayoutManager` 方法创建 `swing.ViewportLayout` 的一个实例, 这个实例用作 JViewport 实例的缺省布局管理器。ViewportLayout 要考虑视口的视图是否实现了 Scrollable 接口并相应地调整视图的大小。有关 Scrollable 接口的更多信息, 请参见接口总结 13-2。

`createViewListener` 方法创建 JViewport.ViewListener 的一个实例, 它被添加到这个视口的视图中。ViewListener 类扩展 `java.awt.event.ComponentAdapter` 类, 而且当重新调整视口的视图大小时, 将激发一个状态变化事件。JViewport 的扩展可以重载 `createViewListener` 方法来修改这种简单的行为 (如果需要的话)。

当添加一个组件到 JViewport 的一个实例中 (即设置了它的视图) 时, 把一个组件监听器 (以 JViewport.ViewListener 的一个实例的形式) 添加到这个视图中, 以便在视图改变大小时激发状态变化事件。所以, 重载 `remove` 方法以便从这个视图中删除这个监听器。

(3) 属性/访问方法

```
public Dimension getExtentSize ()
public final Insets getInsets ()
public Component getView ()
public Point getViewPosition ()
public Rectangle getViewRect ()
public Dimension getViewSize ()

public void setBackingStoreEnabled (boolean)
public final void setBorder (Border)
public void setExtentSize (Dimension)
public void setView (Component)
public void setViewPosition (Point)
public void setViewSize (Dimension)

public boolean isBackingStoreEnabled ()
public boolean isOptimizedDrawingEnabled ()
```

上面所列的这些方法是表 13-1 所列的 JViewport 属性的访问方法。

JViewport 是唯一不允许设置边框的 Swing 组件。我们认为允许为 JViewport 的实例设置边框将使视口的几何图形太复杂, 其结果将使子类化 JViewport 变得很困难。为了确保不能为视口设置边框, JViewport 类从 JComponent 中重载两个方法作为 `final`[⊖], 它们是 `setBorder()` 和 `getInsets()`。`setBorder` 方法弹出一个错误信息, 指出不能为视口设置边框, 而重载 `getInsets` 方法以返回一个为 (0, 0, 0, 0) 的边衬。如果 JViewport 需要一个边框, 则可以把这个视口放到一个面板中, 而这个面板有一个边框。

`setBackingStore` 方法控制 JViewport 的实例是否使用屏外缓存来绘制它们的视图。当视图向一个方向滚动时, 使用屏外缓存可以有效地改进性能。`isBackingStoreEnabled` 方法返回一个 boolean 值, 该值指示当前是否允许使用双缓存。

(4) 绘制/删除/再成形

```
public void paint (Graphics)
public void repaint (long time, int x, int y, int w, int h)
public void reshape (int x, int y, int w, int h)
```

⊖ `final` 方法不能被子类所重载。

因为在 JViewport 的 `backingStoreEnabled` 属性被设置为 `true` 时, JViewport 使用一个屏外缓存, 所以 JViewport 重载 `paint` 方法以便在需要时使用屏外缓存。JViewport 还重载 `repaint()` 方法以使用更有效的方式来进行重新绘制。如果改变了宽度或高度, 则重载 `reshape` 方法来激发一个状态变化事件。

(5) 视图坐标/滚动

```
public void scrollRectToVisible (Rectangle)
public Dimension toViewCoordinates (Dimension)
public Point toViewCoordinates (Point)
```

`scrollRectToVisible` 方法滚动视口的视图中的指定矩形, 以便这个矩形被包含在视口内。参见 13.1.2 节“使用 `scrollRectToVisible` 方法”, 那里有使用 `scrollRectToVisible` 的一个例子。

可以使用 `toViewCoordinates` 的方法来返回 JViewport 扩展的一个点或大小, 这个扩展支持逻辑 (与像素相比) 滚动。这些方法的 JViewport 实现返回与传送给它们的相同的点或大小。

(6) 可访问性和插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public String paramString ()
```

`getAccessibleContext` 方法在 `Accessible` 接口中被定义, 并返回 `AccessibleContext` 的一个实例。`AccessibleContext` 的实例提供有关组件可访问性的信息。

`paramString` 方法返回一个代表这个组件的字符串。

13.1.6 AWT 兼容

AWT 不提供与 JViewport 相似的组件。

13.2 JScrollPane

`JScrollPane` 类是一个轻量 Swing 容器, 它包含一个带可选滚动条的视口和行头部或列头部。

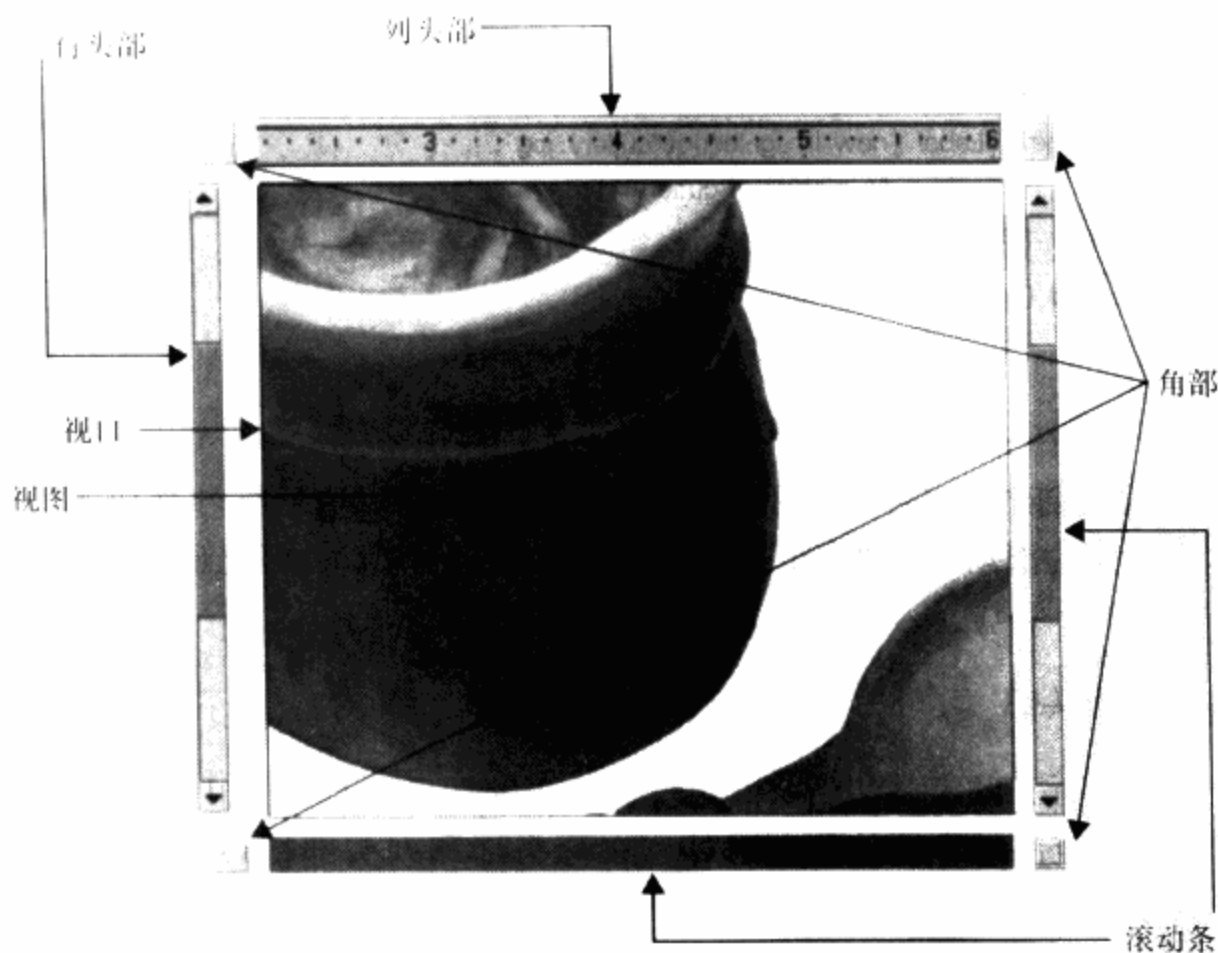


图 13-6 JScrollPane 组件

图 13-6 图解说明了滚动窗格组件。

滚动条被用来滚动视口的视图和行头部/列头部。可以为每个滚动条单独设置滚动条显示策略。要了解 JScrollPane 属性的一个列表，请参见 13.2.3 节“JScrollPane 属性”。

行头部垂直滚动以便在垂直方向上跟踪视口的滚动。同理，列头部水平滚动以便在水平方向上跟踪视口的滚动。在许多情况下（从电子表格的单元指示器到文本编辑器的标尺），头部都是很有用的。

缺省情况下，角部（即头部和滚动条之间的空区域）包含 JPanel 的一个实例，它的背景颜色与这个滚动窗格的背景颜色相同。可以指定组件来替代缺省的角部面板。

可以在构造时使用 JScrollPane.setViewport 方法指定在滚动窗格中显示的视口，而且可以用 JScrollPane.setViewportBorder 方法把一个边框放在滚动窗格的视口周围。

图 13-7 所示的小应用程序中的滚动窗格包含一个带图标标签，这个图标比滚动窗格视口的可见部分还大。

例 13-5 列出了图 13-7 所示小应用程序的代码。

例 13-5 使用 JScrollPane



图 13-7 使用 JScrollPane

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JLabel view = new JLabel (new ImageIcon ("cutlery.jpg",
            "A picture of cutlery"));
        JScrollPane sp = new JScrollPane (view);
        contentPane.add (sp);
    }
}
```

这个小应用程序创建 JLabel 的实例和 JScrollPane 的实例。这个标签被传送给 JScrollPane 构造方法，指示它将作为滚动窗格视口的视图。然后，把滚动窗格添加到这个小应用程序的内容窗格中。

13.2.1 滚动窗格的头部

滚动窗格的可选行头部组件和列头部组件都是 JViewport 的实例，由 JScrollPane 类提供的缺省的头部视口可以由定制版本所替代。另外，在行头部和列头部视口中显示的组件（又称视图）是可设置的。

1. 头部视图

在行头部和列头部显示的视图分别由 JScrollPane 的 setRowHeaderView () 方法和 setColumn-

HeaderView () 方法来指定。任何类型的组件都可以被指定作为滚动窗格的行头部或列头部的视图，但是绝大多数情况下都是使用带图像图标的 JLabel 实例作为行头部或列头部的视图。

图 13-8 所示的小应用程序说明设置一个滚动窗格的行头部视图和列头部视图。这个小应用程序的滚动窗格有头部视图，它们是包含图标（即标尺图像）的标签。

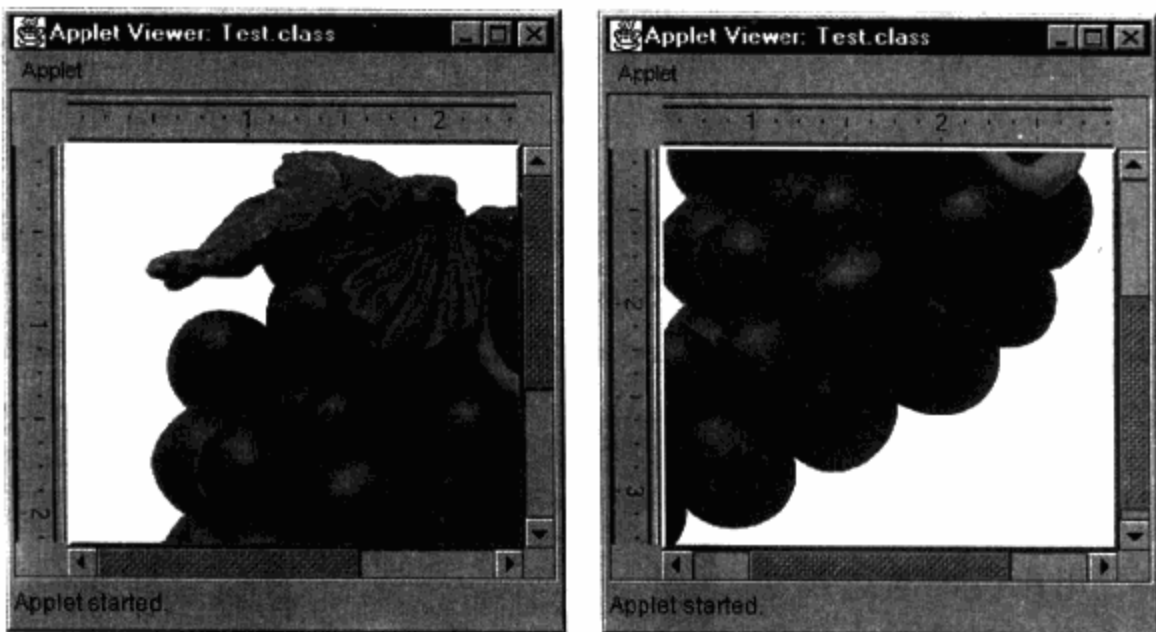


图 13-8 JScrollPane 头部视图

例 13-6 列出了图 13-8 所示小应用程序的代码。

例 13-6 为滚动窗格的头部指定视图

```
import java.awt.*;
import javax.swing.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JLabel columnHeaderView = new JLabel (
            new ImageIcon ("horizontalRuler.jpg",
                "horizontal ruler"));
        JLabel rowHeaderView = new JLabel (
            new ImageIcon ("verticalRuler.jpg",
                "vertical ruler"));
        JLabel view = new JLabel (
            new ImageIcon ("grapes.jpg",
                "grapes"));
        JScrollPane sp = new JScrollPane (view);
        sp.setColumnHeaderView (columnHeaderView);
        sp.setRowHeaderView (rowHeaderView);
        sp.setViewportBorder (
            BorderFactory.createRaisedBevelBorder ());
        contentPane.add (sp);
    }
}
```

这个小应用程序创建 JLabel 的两个实例，这两个实例带有表示水平标尺和垂直标尺的图像。分别调用 JScrollPane.setRowHeaderView () 和 JScrollPane.setColumnHeaderView () 来把这些标签指定为行视图和列视图。

这个小应用程序还指定一个突出的雕刻边框作为这个视口的边框。从类总结 13-1 中我们知道，不能为 Swing 视口直接指定边框，但是 JScrollPane 类提供了一个 setViewportBorder 方法，

这个方法在视口周围包裹一个边框。

注意 行头部视口和列头部视口是不可见的，除非为它们指定了视图。这可以从图 13-7 所示的小应用程序和图 13-8 所示的小应用程序的差别中很明显地看出来。

2. 头部视口

JScrollPane 类除提供行头部视图和列头部视图的访问方法外，还提供了这些视口的访问方法。可以用 JScrollPane.setRowHeader 和 JScrollPane.setColumnHeader 方法来设置这些视口，可以通过调用 JScrollPane.getRowHeader 和 JScrollPane.getColumnHeader 方法来获取对这些视口的引用。

图 13-9 所示的小应用程序包含一个滚动窗格，用户可以通过拖动这个滚动窗格的头部来滚动这个滚动窗格。左图显示这个小应用程序的开始状态，此时光标处于拖动列头部的位上。右图显示在列头部（和这个视口的视图）被从左向右拖动后小应用程序的样子。这个小

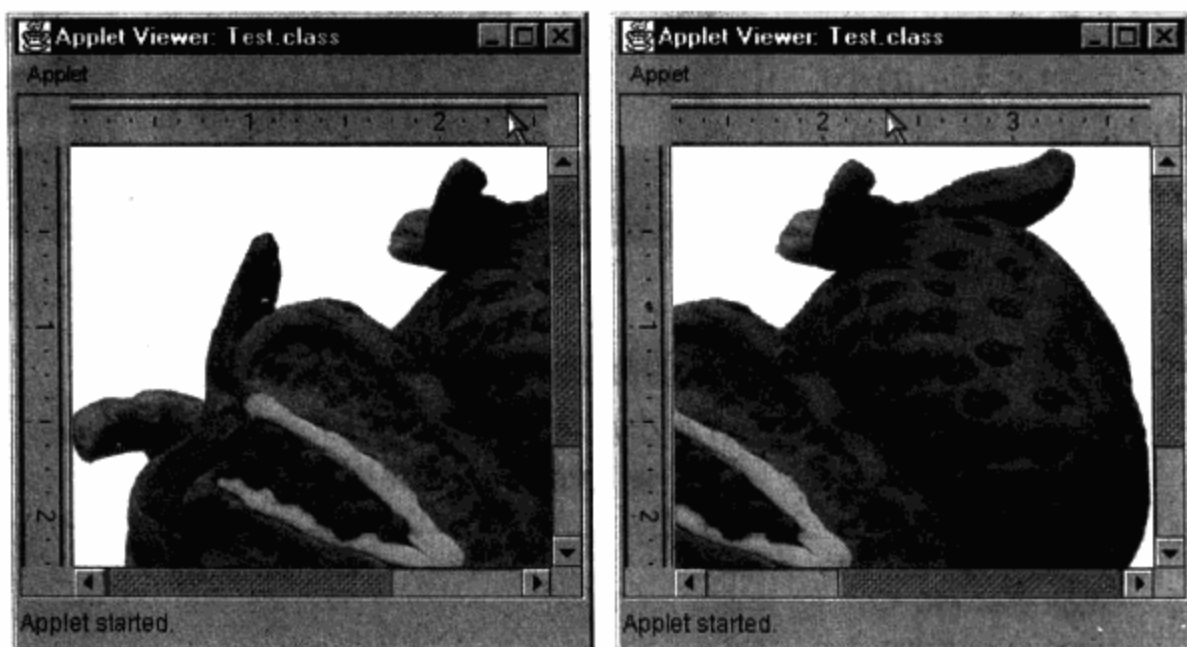


图 13-9 JScrollPane 头部视口

应用程序通过创建定制的行头部视口和列头部视口，并把一个定制监听器添加到这些视口中，来实现头部的拖动。

这个小应用程序创建一个 JScrollPane 实例、两个 JLabel 实例，和两个 JViewport 实例。这两个视口指定作为这个滚动窗格的行头部和列头部，两个标签分别指定作为这两个视口的视图。

```
public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JViewport columnHeaderViewport = new JViewport ();
        JViewport rowHeaderViewport = new JViewport ();

        JLabel columnHeaderView = new JLabel (
            new ImageIcon ("horizontalRuler.jpg"));
        JLabel rowHeaderView = new JLabel (
            new ImageIcon ("verticalRuler.jpg"));
        ...
        // headers must be set before header views
        sp.setColumnHeader (columnHeaderViewport);
        sp.setRowHeader (rowHeaderViewport);

        sp.setColumnHeaderView (columnHeaderView);
        sp.setRowHeaderView (rowHeaderView);
        ...
    }
}
```

注意 头部视口必须在它们的视图前被指定。如果在这些视口前指定这些视图，则这些视图将添加到由滚动窗格创建的缺省视口中。当这些定制视口替代滚动窗格的缺省视口时，它们将不能与这些视图相关联。

这个小应用程序还创建两个 `HeaderViewDragListener` 实例，它们被添加到这两个视口中作为鼠标监听器和鼠标移动监听器。

```
...
HeaderViewDragListener verticalHeaderListener =
    new HeaderViewDragListener (sp,
        SwingConstants.VERTICAL);

HeaderViewDragListener horizontalHeaderListener =
    new HeaderViewDragListener (sp,
        SwingConstants.HORIZONTAL);

columnHeaderViewport.addMouseListener (
    horizontalHeaderListener);
columnHeaderViewport.addMouseMotionListener (
    horizontalHeaderListener);

rowHeaderViewport.addMouseListener (
    verticalHeaderListener);
rowHeaderViewport.addMouseMotionListener (
    verticalHeaderListener);
...
```

`HeaderViewDragListener` 类扩展 `java.awt.event.MouseAdapter` 类，并实现 `java.awt.event.MouseMotionListener` 接口和 `SwingConstants` 接口。

用两个参数来构造 `HeaderViewDragListener` 的实例，这两个参数是对滚动窗格的一个引用和与这些实例相关联头部视口的方向。当在头部视口中发生一个鼠标按下事件时，就记录鼠标被按下的位置。

```
class HeaderViewDragListener extends MouseAdapter
    implements MouseMotionListener,
        SwingConstants {

    private Point last = new Point ();
    private JScrollPane scrollpane;
    private int orientation;

    public HeaderViewDragListener (JScrollPane sp, int orient) {
        scrollpane = sp;
        orientation = orient;
    }

    public void mousePressed (MouseEvent e) {
        last.x = e.getPoint ().x;
        last.y = e.getPoint ().y;
    }

    public void mouseMoved (MouseEvent e) {
    }
    ...
}
```

当在一个头部视口中拖动鼠标时，调用 `java.Util.EventObject.getSource()` 来获得对发生事件的头部视口的一个引用。调用 `JScrollPane.getViewport()` 来获得与滚动窗格相关联的主视口。接着，调用 `JViewport` 的 `getViewSize` 和 `getExtentSize` 方法来获取主视口的大小和可见部分。

调用 `MouseEvent.getPoint()` 来获取拖动事件的位置，并计算当前的拖动位置与最后鼠标事件的位置之间的偏移。然后，调用 `JViewport.getViewPosition()` 来获得主视口视图的当前位置。

```
...
public void mouseDragged (MouseEvent e) {
    JViewport headerViewport = (JViewport) e.getSource ();
```

```

JViewport scrollpaneViewport = scrollpane.getViewport ();
Dimension viewSize = scrollpaneViewport.getViewSize (),
    extent = scrollpaneViewport.getExtentSize ();

Point drag = e.getPoint ();
Point offset = new Point (drag.x-last.x, drag.y-last.y);
Point headerPosition = new Point (), viewportPosition;
viewportPosition = scrollpaneViewport.getViewPosition ();
...

```

通过从主视口的当前位置减去当前的拖动位置与最后的鼠标事件位置之间的偏移来计算下一个拖动位置。如果下一个拖动位置在主视口的边界内，则头部视口和主视口的位置都要被更新。

```

...
if (orientation == HORIZONTAL) {
    int nextX = viewportPosition.x - offset.x;
    int rightEdge = extent.width + nextX;

    if (nextX > 0 && rightEdge < viewSize.width) {
        headerPosition.x = nextX;
        viewportPosition.x = nextX;
    }
}

if (orientation == VERTICAL) {
    int nextY = viewportPosition.y - offset.y;
    int bottomEdge = extent.height + nextY;

    if (nextY > 0 && bottomEdge < viewSize.height) {
        headerPosition.y = nextY;
        viewportPosition.y = nextY;
    }
}

headerViewport.setViewPosition (headerPosition);
scrollpaneViewport.setViewPosition (viewportPosition);

last.x = drag.x;
last.y = drag.y;
}
}

```

例 13-7 列出了图 13-9 所示小应用程序的完整代码。

例 13-7 设置头部视口

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JViewport columnHeaderViewport = new JViewport ();
        JViewport rowHeaderViewport = new JViewport ();

        JLabel columnHeaderView = new JLabel (
            new ImageIcon ("horizontalRuler.jpg"));
        JLabel rowHeaderView = new JLabel (
            new ImageIcon ("verticalRuler.jpg"));
        JLabel view = new JLabel (
            new ImageIcon ("strawberry.jpg"));
    }
}

```

```

JScrollPane sp = new JScrollPane (view);
sp.setToolTipText (
    "Drag the headers to drag the picture!");
HeaderViewDragListener verticalHeaderListener =
    new HeaderViewDragListener (sp,
        SwingConstants.VERTICAL);
HeaderViewDragListener horizontalHeaderListener =
    new HeaderViewDragListener (sp,
        SwingConstants.HORIZONTAL);
columnHeaderViewport.addMouseListener (
    horizontalHeaderListener);
columnHeaderViewport.addMouseMotionListener (
    horizontalHeaderListener);
rowHeaderViewport.addMouseListener (
    verticalHeaderListener);
rowHeaderViewport.addMouseMotionListener (
    verticalHeaderListener);

// headers must be set before header views
sp.setColumnHeader (columnHeaderViewport);
sp.setRowHeader (rowHeaderViewport);
sp.setColumnHeaderView (columnHeaderView);
sp.setRowHeaderView (rowHeaderView);
contentPane.add (sp);
}
}

class HeaderViewDragListener extends MouseAdapter
    implements MouseMotionListener,
        SwingConstants {
    private Point last = new Point ();
    private JScrollPane scrollpane;
    private int orientation;

    public HeaderViewDragListener (JScrollPane sp, int orient) {
        scrollpane = sp;
        orientation = orient;
    }

    public void mousePressed (MouseEvent e) {
        last.x = e.getPoint ().x;
        last.y = e.getPoint ().y;
    }

    public void mouseMoved (MouseEvent e) {
    }

    public void mouseDragged (MouseEvent e) {
        JViewport headerViewport = (JViewport) e.getSource ();
        JViewport scrollpaneViewport = scrollpane.getViewport ();
        Dimension viewSize = scrollpaneViewport.getViewSize (),
            extent = scrollpaneViewport.getExtentSize ();

        Point drag = e.getPoint ();
        Point offset = new Point (drag.x-last.x, drag.y-last.y);
        Point headerPosition = new Point (), viewportPosition;
        viewportPosition = scrollpaneViewport.getViewPosition ();

```

```

if (orientation == HORIZONTAL) {
    int nextX = viewportPosition.x - offset.x;
    int rightEdge = extent.width + nextX;
    if (nextX > 0 && rightEdge < viewSize.width) {
        headerPosition.x = nextX;
        viewportPosition.x = nextX;
    }
}
if (orientation == VERTICAL) {
    int nextY = viewportPosition.y - offset.y;
    int bottomEdge = extent.height + nextY;
    if (nextY > 0 && bottomEdge < viewSize.height) {
        headerPosition.y = nextY;
        viewportPosition.y = nextY;
    }
}
headerViewport.setViewPosition (headerPosition);
scrollpaneViewport.setViewPosition (viewportPosition);
last.x = drag.x;
last.y = drag.y;
}
}

```

13.2.2 滚动窗格的角部

JScrollPane 的每个实例都有四个角部，它们代表滚动窗格的滚动条与头部之间的空间。JScrollPane 提供了一个 setCorner 方法，它把一个组件插入到这四个角部中的任何一个。这个方法带一个代表角部的字符串和一个 java.awt.Component 实例。这个字符串是由 ScrollPaneConstants 接口定义四个常量中的一个常量。有关 ScrollPaneConstants 接口的情况，请参见接口总结 13-1。

缺省情况下，一个滚动窗格的角部被 JPanel 的不透明的实例所占据，这些实例用从 UIManager.getColor(control) 返回的颜色来绘制它们的背景。有关使用 UIManager 来访问缺省值的更多信息，请参见 7.1.3 节“UI 管理器”。

图 13-10 所示的小应用程序把组件插入到滚动窗格所包含的每个角部中。

例 13-8 列出了图 13-10 所示小应用程序的代码。

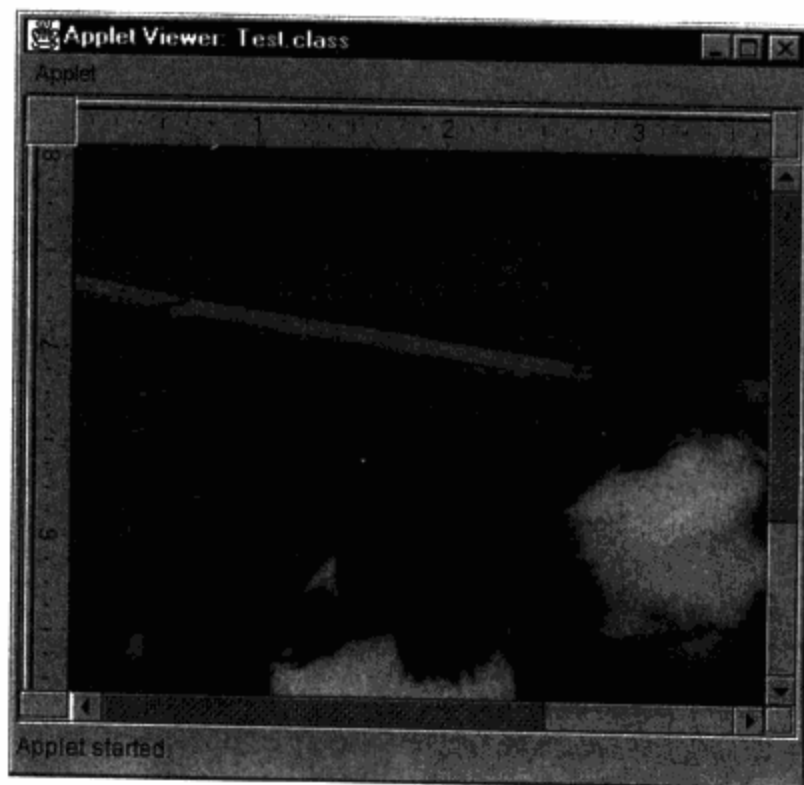


图 13-10 为滚动窗格的角部指定组件

例 13-8 指定滚动窗格的角部的组件

```

import java.awt.*;
import javax.swing.*;

```

```

import javax.swing.border.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JLabel columnHeaderView = new JLabel (
            new ImageIcon ("horizontalRuler.jpg"));
        JLabel rowHeaderView = new JLabel (
            new ImageIcon ("verticalRuler.jpg"));
        JLabel view = new JLabel (
            new ImageIcon ("anjinAndMariko.gif"));

        JScrollPane sp = new JScrollPane (view);

        JPanel corners [] = {
            new JPanel (), new JPanel (),
            new JPanel (), new JPanel ()
        };

        String cornerConstants [] = {
            ScrollPaneConstants.UPPER_LEFT_CORNER,
            ScrollPaneConstants.LOWER_LEFT_CORNER,
            ScrollPaneConstants.UPPER_RIGHT_CORNER,
            ScrollPaneConstants.LOWER_RIGHT_CORNER,
        };

        Border border = BorderFactory.createEtchedBorder ();

        for (int i = 0; i < corners.length; ++i) {
            corners [i].setBorder (border);
            sp.setCorner (cornerConstants [i], corners [i]);
        }

        sp.setColumnHeaderView (columnHeaderView);
        sp.setRowHeaderView (rowHeaderView);
        contentPane.add (sp);
    }
}

```

这个小应用程序创建四个 JPanel 实例，把它们作为角部组件使用。每个面板都配备了一个从 Swing 边框库获得的蚀刻边框。JScrollPane.setCorner 方法把每个面板插入到这个小应用程序所包含的滚动窗格的每个角部中。

Swing 提示

滚动窗格角部的缺省组件

应该注意到，例 13-8 所列的小应用程序不需要为角部组件创建 JPanel 的实例，因为缺省时，角部已经由 JPanel 的实例所占据。例 13-8 所列的小应用程序中的 for 循环应该像下面这样写：

```

for (int i = 0; i < corners.length; ++i) {
    JComponent panel =
        (JComponent) sp.getCorner (cornerConstants [i]);
    panel.setBorder (border);
}

```

...，小应用程序完全可以免去创建面板。然而，由于 JScrollPane.getCorner 方法返回 java.awt.Component 的一个实例，所以上面所列的 for 循环依赖一个实现细节（即缺省的角部组件

是 JPanel 的实例), 而不是依赖由 JScrollPane 类提供的公共方法。依赖一个类的实现细节是一件很冒险的事, 因为实现细节常改变, 而一个类的 API 要稳定的多。

组件总结 13-2 总结了 JScrollPane 组件。

组件总结 13-2 JScrollPane

模型	——
UI 代表:	<code>javax.swing.plaf.basic.BasicScrollPaneUI</code>
绘制器:	——
编辑器:	——
激发的事件:	<code>PropertyChangeEvent</code>
替换:	<code>java.awt.ScrollPane</code>
类图:	

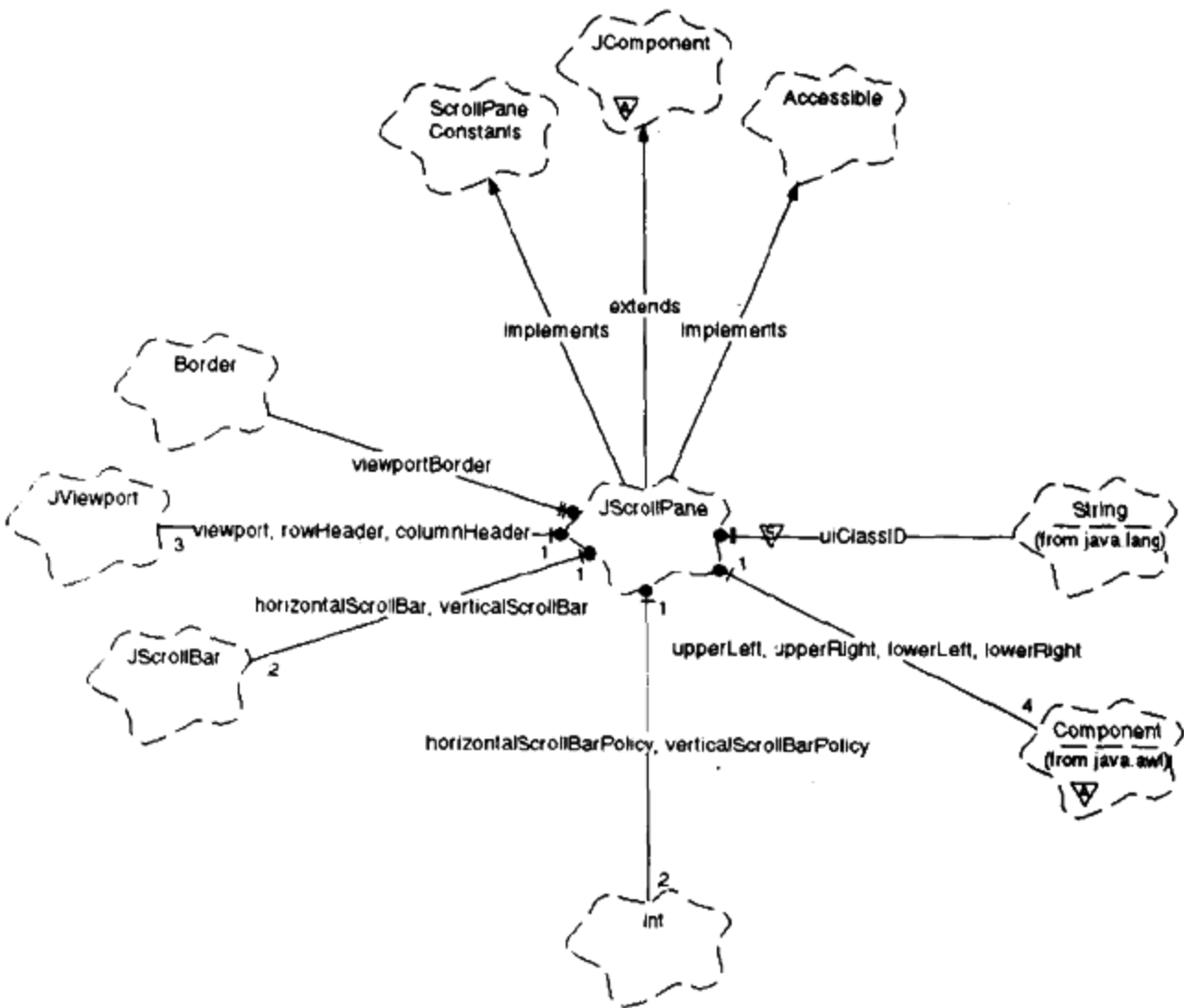


图 13-11 JScrollPane 类图

JScrollPane 扩展 JComponent 并实现 ScrollPaneConstants 接口和 Accessible 接口。
JScrollPane 的实例维护对滚动窗格的主视口、行头部视口和列头部视口的 protected 引用。
JScrollPane 类除维护占据滚动窗格角部的组件外, 还维护对滚动窗格的水平和垂直滚动条的 protected 引用, 并维护管理滚动条显示的策略。

13.2.3 JScrollPane 属性

表 13-2 列出了由 JScrollPane 类维护的属性。

表 13-2 JScrollPane 的属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
columnHeader	JViewport	B	SG	null
columnHeaderView	Component	B	S ^④	null
corner	Component	IB	SG	null
horizontalScrollbar	JScrollbar	B	SG	——
horizontalScrollbarPolicy	int	B	CSG	按需要 ^⑥
rowHeader	JViewport	B	SG	null
rowHeaderView	Component	B	S ^④	null
verticalScrollbar	JScrollbar	B	SG	——
verticalScrollbarPolicy	int	B	CSG	按需要 ^⑦
viewport	JViewport	B	SG	JViewport
viewportBorder	Border	B	SG	null
viewportView	Component	B	CSG	null

① B = 关联的 (激发 PropertyChangedEvent) / C = 受约束的 / I = 索引的 / S = 简单的 / Ch = 激发 ChangeEvent
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ getColumnHeader, getView () 返回列的头部视图
④ getRowHeader, getView () 返回行的头部视图
⑤ I&F = 与界面样式有关
⑥ JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
⑦ JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED

columnHeader——用于列头部的一个 JViewport 实例

columnHeaderView——一个 Component 实例，作为列头部视口的视图。

corner——在滚动窗格的角部中显示的组件。滚动窗格的角部用下述字符串之一来指定：

- ScrollPaneConstants.UPPER_LEFT_CORNER
- ScrollPaneConstants.LOWER_LEFT_CORNER
- ScrollPaneConstants.UPPER_RIGHT_CORNER
- ScrollPaneConstants.LOWER_RIGHT_CORNER

horizontalScrollbar——滚动窗格使用的水平滚动条。这个滚动条是 JScrollPane.JScrollbar 的一个实例，是 JScrollbar 的一个扩展，这个扩展考虑这个滚动窗格所包含的视图是否实现了 Scrollable 接口。

horizontalScrollbarPolicy——用于确定显示水平滚动条的环境的策略。这个策略用下述 integer 值之一来指定：

- ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED
- ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER
- ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS

rowHeader——用于行头部的一个 JViewport 实例。

rowHeaderView——Component 的一个实例，它被用作行头部视口的视图。

verticalScrollbar——滚动窗格使用的垂直滚动条。这个滚动条是 JScrollPane.JScrollbar 的一个实例，是 JScrollbar 的一个扩展，扩展考虑滚动窗格所包含的视图是否实现了 Scrollable 接口。

verticalScrollbarPolicy——用于确定显示垂直滚动条的环境的策略。这个策略用下述 integer 值之一来指定：

- ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED
- ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER
- ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS

viewport——JViewport 的一个实例，用于显示被滚动窗格滚动的组件。

viewportBorder——视口的边框

viewportView——在滚动窗格的视口中显示的组件。

13.2.4 JScrollPane 事件

JScrollPane 类在修改其关联属性时将激发属性变化事件。要了解 JScrollPane 的关联属性列表，请参见表 13-2。

13.2.5 JScrollPane 类总结

JScrollPane 实现 ScrollPaneConstants 接口，这个接口定义接口总结 13-1 中所列的字符串和 integer 常量。

接口总结 13-1 JScrollPaneConstants

1. 常量

```
public static final String COLUMN_HEADER
public static final String HORIZONTAL_SCROLLBAR
public static final int HORIZONTAL_SCROLLBAR_ALWAYS
public static final int HORIZONTAL_SCROLLBAR_AS_NEEDED
public static final int HORIZONTAL_SCROLLBAR_NEVER
public static final String HORIZONTAL_SCROLLBAR_POLICY
public static final String LOWER_LEFT_CORNER
public static final String LOWER_RIGHT_CORNER
public static final String ROW_HEADER
public static final String UPPER_LEFT_CORNER
public static final String UPPER_RIGHT_CORNER
public static final int VERTICAL_SCROLLBAR
public static final int VERTICAL_SCROLLBAR_ALWAYS
public static final int VERTICAL_SCROLLBAR_NEVER
public static final String VERTICAL_SCROLLBAR_POLICY
public static final String VIEWPORT
```

ScrollPaneConstants 类定义上述常量列表，这些常量代表滚动条显示策略和滚动窗格中的组件。大多数由 ScrollPaneConstants 接口定义的常量被 JScrollPane、ScrollPaneLayout 和滚动窗格的 UI 代表在内部使用。然而，下面所列的常量由开发人员用来指定角部组件、水平和垂直滚动条显示策略。

2. 方法

(1) 滚动条显示策略

```
public static final int HORIZONTAL_SCROLLBAR_ALWAYS
public static final int HORIZONTAL_SCROLLBAR_AS_NEEDED
public static final int HORIZONTAL_SCROLLBAR_NEVER
public static final int VERTICAL_SCROLLBAR_ALWAYS
public static final int VERTICAL_SCROLLBAR_AS_NEEDED
public static final int VERTICAL_SCROLLBAR_NEVER
```

上面所列的 integer 常量指定滚动窗格的滚动条的显示策略。当构造 JScrollPane 时或在构造

后调用 JScrollPane 的 setHorizontalScrollBarPolicy 和 setVerticalScrollBarPolicy 方法（这两个方法都必须以上面所列的常量之一作为参数）可以指定滚动条显示策略。

如果没有显式地设置一个滚动条策略，那么缺省的策略是：水平滚动条为 HORIZONTAL_SCROLLBAR_AS_NEEDED，垂直滚动条为 VERTICAL_SCROLLBAR_AS_NEEDED。

(2) 角部常量

```
public static final String LOWER_LEFT_CORNER
public static final String LOWER_RIGHT_CORNER
public static final String UPPER_LEFT_CORNER
public static final String UPPER_RIGHT_CORNER
```

把上述所列的字符串常量传送给 JScrollPane 的方法 setCorner 和 getCorner 中，以便指定一个特定的角部。

类总结 13-2 列出了 JScrollPane 的 public 和 protected 变量

类总结 13-2 JScrollPane

扩展：JComponent

实现：ScrollPaneConstants, javax.accessibility.Accessible

1. 构造方法

```
public JScrollPane ()
public JScrollPane (int vsbPolicy, int hsbPolicy)
public JScrollPane (Component view)
public JScrollPane (Component view, int vsbPolicy, int hsbPolicy)
```

JScrollPane 提供四个构造方法。传送给 JScrollPane 构造方法的 integer 值代表垂直和水平滚动条显示策略（按构造方法中所写的顺序）。传送给构造方法的组件被用作视口的视图，即它是由滚动窗格滚动的组件。

无参数的构造方法构造一个滚动窗格，它用 null 组件作为视口的视图，滚动条显示策略按需要显示水平和垂直滚动条。

2. 方法

(1) 创建方法

```
public JScrollBar createHorizontalScrollBar ()
public JScrollBar createVerticalScrollBar ()
protected JViewport createViewport ()
```

与大多数 Swing 组件一样，JScrollPane 提供创建其子组件的 create... 方法。然而，与大多数 Swing 组件不同，JScrollPane 实现的创建其滚动条的方法是 public 方法而不是 protected 方法，因为这些方法是从滚动窗格的 UI 代表中调用的。

createHorizontalScrollBar 和 createVerticalScrollBar 方法都返回 JScrollPane.ScrollBar 的实例，这些实例是 JScrollPane 的一个扩展，它考虑滚动窗格的视图是否实现了 Scrollable 接口。有关 Scrollable 接口的更多信息，请参见接口总结 13-2。

对实现 create... 方法的所有 Swing 组件来说，可以在扩展类中重载这些方法，以便使用定制组件替代缺省的子组件。

(2) 属性访问方法

```
public JViewport getColumnHeader ()
public Component getCorner (String)
public JScrollBar getHorizontalScrollBar ()
```

```

public void setHorizontalScrollBar (JScrollBar)
public int getHorizontalScrollBar Policy ()
public JViewport getRowHeader ()
public JScrollBar getVerticalScrollBar ()
public int getVerticalScrollBarPolicy ()
public JViewport getViewport ()
public Border getViewportBorder ()
public void getViewportBorderBounds (Border)

public void setColumnHeader (JViewport)
public void setColumnHeaderView (Component)
public void setCorner (String, Component)
public void setHorizontalScrollBarPolicy (int)
public void setRowHeader (JViewport)
public void setRowHeaderView (Component)
public void setVerticalScrollBar (JScrollBar)
public void setVerticalScrollBarPolicy (int)
public void setViewport (JViewport)
public void setViewportBorder (Border)
public void setViewportView (Component)

public boolean isOpaque ()
public boolean isValidateRoot ()

```

上面所列的这些方法是 JScrollPane 属性的访问方法。上面所列的方法几乎都在本章前面的小应用程序中使用过。

与 JRootPane 和 JTextField 一样, JScrollPane 重载 isValidateRoot 方法来返回 true。如果一个 Swing 容器从 isValidateRoot 中返回 true, 则在该容器所包含的组件之一上调用 revalidate () 将引起这个容器使它所包含的所有组件有效。使一个 Swing (或 AWT) 容器中的所有组件有效导致这些组件被布局和重新绘制。要了解使用 revalidate 方法的例子, 请参见 13.3 节 “Scrollable 接口” 和 13.2.4 节 “JScrollPane 事件”。

如果 JScrollPane 的视口比视口所包含的视图小, 那么 JScrollPane 重载 isOpaque 方法来返回 true。如果这个视口比它的视图大, 则滚动窗格是透明的, 即在滚动窗格下面的背景将透过这个视口的视图和滚动窗格之间的空隙显示出来。

(3) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public ScrollPaneUI getUI ()
public String getUIClassID ()
public void setUI (ScrollPaneUI)
public void updateUI ()

```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

不透明滚动窗格与透明滚动窗格的比较

图 13-2 所示的小应用程序图解说明了不透明滚动窗格与透明滚动窗格的比较。这个小应用程序包含两个滚动窗格, 左边的滚动窗格有一个透明的背景, 而右边的滚动窗格是不透明的。左边的滚动窗格是透明的, 因为它的视图没有完全填满滚动窗格的视口。相反, 右边的滚动窗格是不透明的, 因为它的视图比滚动窗格的视口大, 因此, 没有空着的空间让背景透过它显示出来。

这个小应用程序把它的内容窗格设置为 CustomContentPane 的一个实例, 这个实例用一幅图像平铺其背景。对两个 JLabel 实例和两个 JScrollPane 实例进行实例化, 把这两个标签分别指定

为这两个滚动窗格视口的视图。然后，把这两个滚动窗格添加到定制的内容窗格中。

```
public class Test extends JApplet {
    public void init () {
        Container contentPane = new CustomContentPane ();
        JLabel view1 = new JLabel (
            new ImageIcon ("gjMedium.gif"));
        JLabel view2 = new JLabel (
            new ImageIcon ("anjinAndMariko.gif"));

        JScrollPane sp1 = new JScrollPane (view1);
        JScrollPane sp2 = new JScrollPane (view2);

        setContentPane (contentPane);
        sp1.setPreferredSize (new Dimension (250, 250));
        sp2.setPreferredSize (new Dimension (250, 250));

        contentPane.add (sp1);
        contentPane.add (sp2);
    }
}
```

例 13-9 列出了图 13-12 所示小应用程序的完整代码。

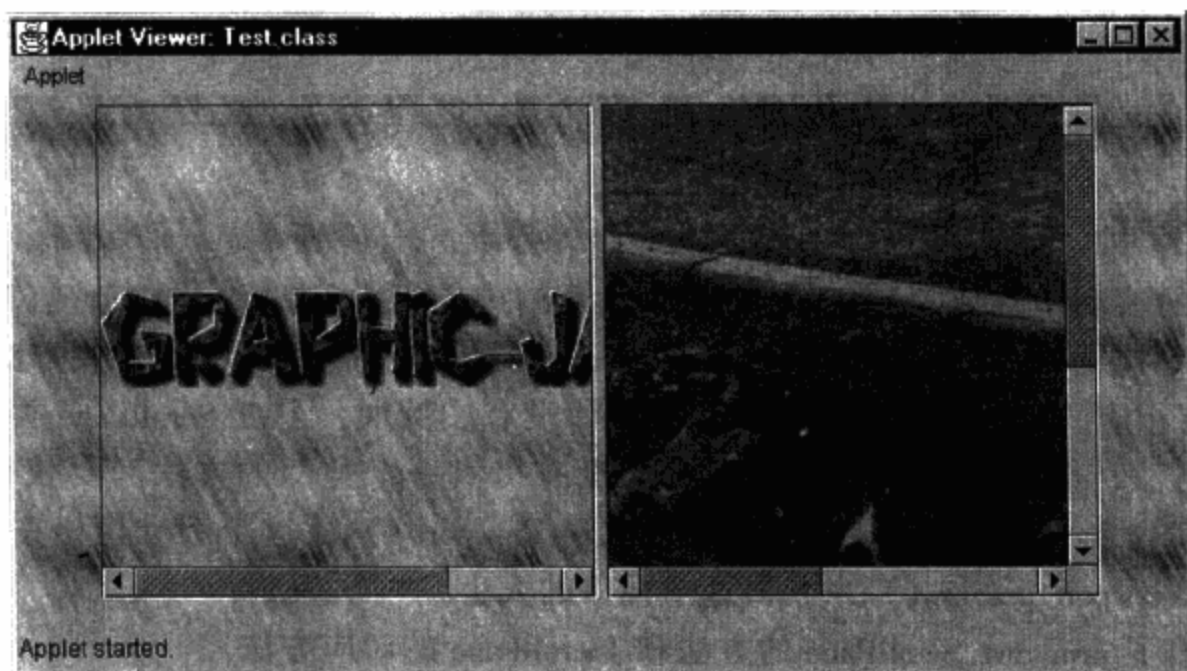


图 13-12 透明与不透明滚动窗格

例 13-9 透明和不透明滚动窗格

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = new CustomContentPane ();
        JLabel view1 = new JLabel (
            new ImageIcon ("gjMedium.gif"));
        JLabel view2 = new JLabel (
            new ImageIcon ("anjinAndMariko.gif"));

        JScrollPane sp1 = new JScrollPane (view1);
        JScrollPane sp2 = new JScrollPane (view2);
```

```
        setContentPane (contentPane);
        sp1.setPreferredSize (new Dimension (250, 250));
        sp2.setPreferredSize (new Dimension (250, 250));

        contentPane.add (sp1);
        contentPane.add (sp2);
    }
}

class CustomContentPane extends JPanel {
    private ImageIcon rain = new ImageIcon ("rain.gif");

    public CustomContentPane () {
        setLayout (new FlowLayout ());
    }

    public void paintComponent (Graphics g) {
        int rainw = rain.getIconWidth ();
        int rainh = rain.getIconHeight ();

        Dimension size = getSize ();

        for (int row = 0; row < size.height; row += rainh)
            for (int col = 0; col < size.width; col += rainw)
                rain.paintIcon (this, g, col, row);
    }
}
```

13.2.6 AWT 兼容

Swing 组件的滚动体系结构与 AWT 的滚动体系结构有很大的差别。AWT 视口是一个逻辑构造，即 AWT 不包括一个视口类。因此，在 `java.awt.ScrollPane` 中实现的有些方法在 swing 的 `JViewport` 类中有相应的方法，但在 `JScrollPane` 中却没有。例如，AWT `ScrollPane` 类提供了一个 `getViewportSize` 方法，但是 Swing 的 `JScrollPane` 却没有提供一个类似的方法。要获得一个 Swing 滚动窗格的视口的大小，必须首先用 `JScrollPane.getViewport()` 从这个滚动窗格中获得这个视口，然后，用 `JViewport.getSize()` 来获得这个视口的大小。

表 13-3 列出了 `java.awt.ScrollPane` 的方法和 `JScrollPane` 的对应方法。

表 13-3 `java.awt.ScrollPane` 的方法和 `JScrollPane` 的对应方法

java.awt.ScrollPane 的方法	JScrollPane 的对应方法
<code>Adjustable getHAdjustable ()</code>	<code>JScrollBar getHorizontalScrollBar ()</code>
<code>int getHScrollbarHeight ()</code>	——
<code>int getScrollbarDisplayPolicy ()</code>	<code>int getHorizontalScrollBarPolicy () /</code> <code>int getVerticalScrollBarPolicy ()</code>
<code>Point getScrollPosition ()</code>	<code>aScrollPane.getViewport ().getViewPosition ()</code>
<code>Adjustable getVAdjustable ()</code>	<code>JScrollBar getVerticalScrollBar ()</code>
<code>Dimension getViewportSize ()</code>	<code>aScrollPane.getViewport ().getSize ()</code>
<code>int getVScrollbarWidth ()</code>	——
<code>void setScrollPosition (int, int)</code>	<code>aScrollPane.getViewport ().setViewPosition (Point)</code>
<code>void setScrollPosition (Point)</code>	——

与 `java.awt.ScrollPane` 的实例相关联的滚动条由作为 `java.awt.Adjustable` 的实例来访问，而 `JScrollPane` 滚动条由作为 `JScrollBar` 的实例来访问。

`java.awt.ScrollPane` 和 `JScrollPane` 的另一个差别是 `JScrollPane` 允许它的滚动条有不同的显示

策略，而 AWT 的 ScrollPane 强迫它的两个滚动条有相同的显示策略。

13.3 Scrollable 接口

任何 Swing 或 AWT 组件都可以在 JScrollPane 的一个实例中滚动。实际上，实现 Scrollable 接口的组件有特殊的滚动需求。下面的 Swing 组件实现 Scrollable 接口：

- Jlist
- JTable
- JTextComponent
- Tree

接口总结 13-2 介绍了 Scrollable 接口定义的五五个方法。

接口总结 13-2 Scrollable

```
public abstract Dimension getPreferredSize() { }
```

对于一个包含在视口中的可滚动组件来说，getPreferredSize 方法返回组件所需的视口的大小。

例如，当把一个列表放在一个滚动容器中时，必须设置这个列表的首选高度以便能容纳可视的行数。虽然 JList 的实例的首选高度是显示所有行所需的高度，但是列表的视口的首选高度是显示这个列表可视行所需的高度。

```
public abstract int getScrollableBlockIncrement (Rectangle visibleRect,
                                                    int orientation, int direction) { }
public abstract int getScrollableUnitIncrement (Rectangle visibleRect,
                                                  int orientation, int direction) { }
```

getScrollableBlockIncrement 应该返回显示一块行或一块列所需的像素数。通常把一个块定义为在视口中当前可见的行数或列数。getScrollableUnitIncrement 应该返回显示视口视图中的下一行或下一列所需的像素数。

getScrollableBlockIncrement 和 getScrollableUnitIncrement 都有一个矩形参数，它代表视图当前可见的部分，还有一个 integer 参数，它代表滚动条的方向，这个值是 SwingConstants.VERTICAL 或是 SwingConstants.HORIZONTAL，另外，还有一个 integer 参数，它代表滚动的方向。滚动方向值小于 0 表示向上或向下滚动，滚动方向值大于 0 表示向左或向右滚动。

图 13-13 所示的小应用程序说明了可滚动组件的单元增量和块增量。

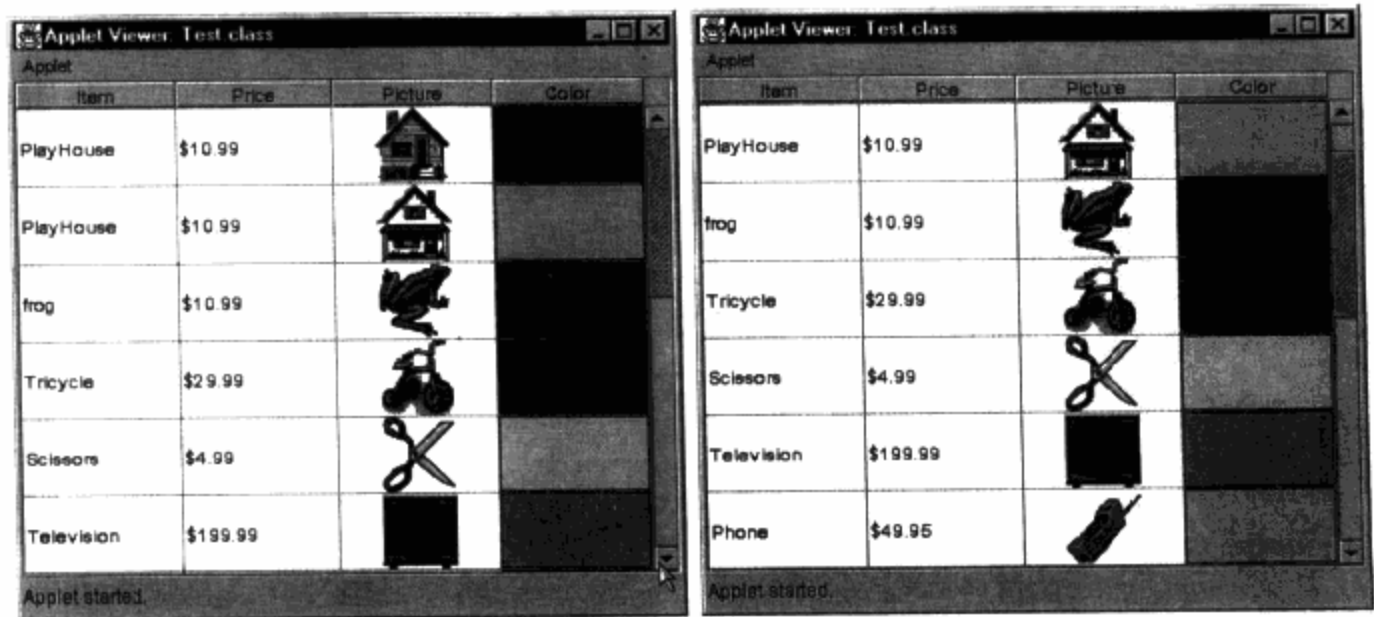


图 13-13 单元增量和块增量

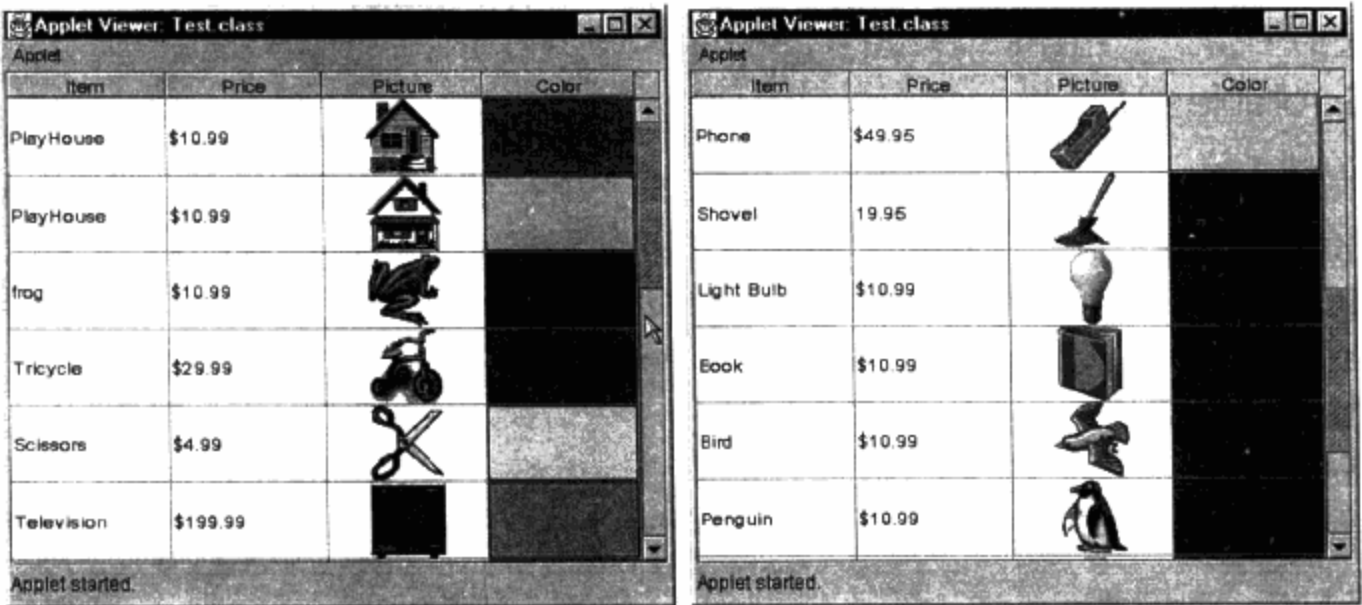


图 13-13 单元增量和块增量（续）

左上图显示了这个小应用程序开始的样子，此时鼠标光标处于滚动条的向下箭头上并准备点击向下箭头，这次点击将引起表格滚动一个单元增量。滚动一个单元增量后，其结果如右上图所示。

左下图也显示这个小应用程序开始时的样子，此时鼠标光标处于滚动条的槽内并准备点击槽，这次点击将引起表格滚动一个块增量。滚动一个块增量后，其结果如右下图所示。

本章没有列出图 13-13 所示的小应用程序，有关“JTable”的详细内容，请参见第 19 章“表格”。

```
public abstract boolean getScrollableTracksViewportHeight ()
public abstract boolean getScrollableTracksViewportWidth ()
```

上面所列的这两个方法都返回 boolean 值，这两个 boolean 值分别指示视口是否应该迫使其视图的宽度与视口的宽度相同，或是否应该迫使其视图的高度与视口的高度相同。

JTextArea 组件实现 getScrollableTracksViewportWidth 以便返回 true（如果文本域允许自动换行的话）。这样就确保了在文本域中显示的文本能自动换行而不会延伸超出视口的右边界。

图 13-14 所示的小应用程序包含一个文本域和一个复选框，这个文本域包含在一个滚动窗格中，这个复选框用来设置文本域的自动换行功能。当设置自动换行为 false 时，如图 13-14 左

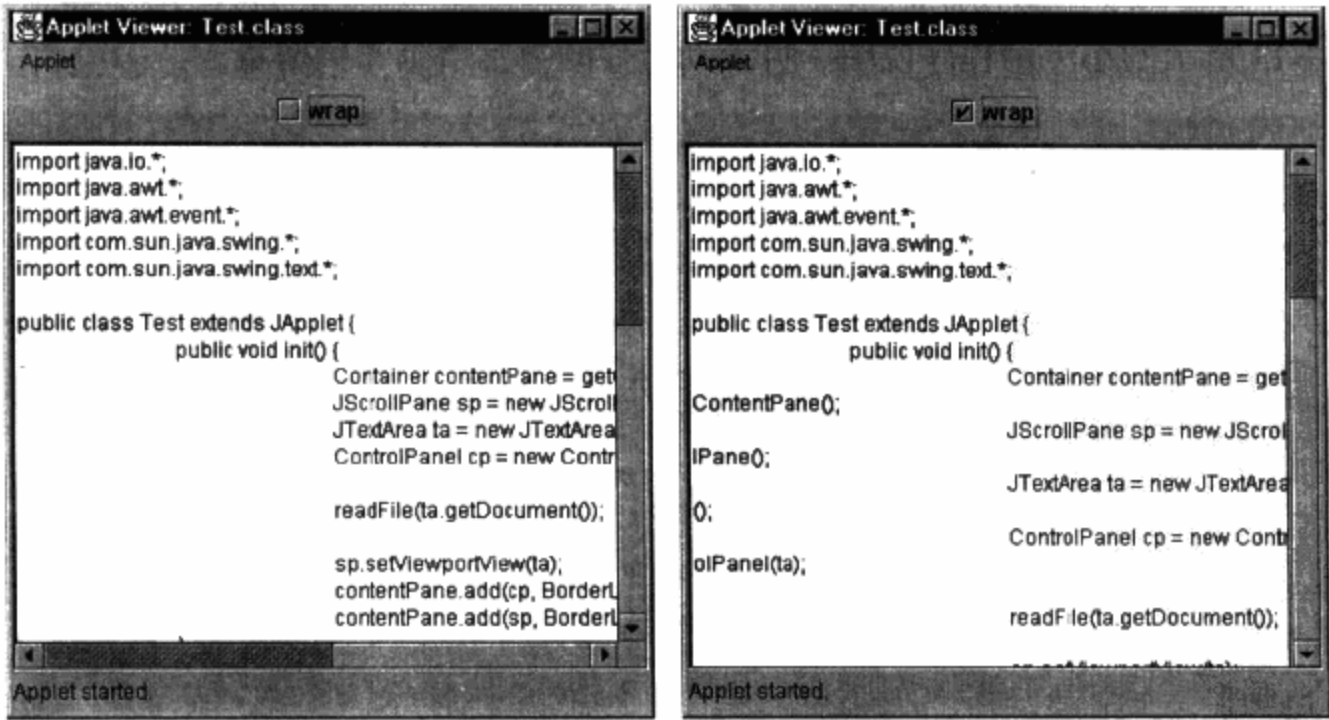


图 13-14 跟踪可滚动组件的视口大小

图所示，文本域的宽度超出了滚动窗格的宽度，从水平滚动条中可以明显地看到。当设置自动换行为 true 时，如图 13-14 右图所示，文本域的宽度等于滚动窗格的宽度，因此不显示水平滚动条。

复选框包含在 ControlPanel 的一个实例中，ControlPanel 是 JPanel 类的一个扩展。当选取了这个复选框时，则允许文本域中的文本自动换行。当没有选取这个复选框时，则不允许文本域中的文本自动换行。

```
class ControlPanel extends JPanel {
    public ControlPanel (final JTextArea ta) {
        final JCheckBox cb = new JCheckBox ("wrap");
        add (cb);
        cb.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                if (cb.isSelected ())
                    textArea.setLineWrap (true);
                else
                    textArea.setLineWrap (false);
            }
        });
    }
}
```

例 13-10 列出了图 13-14 所示的小应用程序的完整代码。有关文本域的更多信息，请参见 22.3 节“JTextArea”。

例 13-10 跟踪视口宽度

```
import java.io. * ;
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.text. * ;

public class Test extends JApplet {
    private JTextArea textArea = new JTextArea ();

    public void init () {
        Container contentPane = getContentPane ();
        readFile ();

        contentPane.add (new ControlPanel (), BorderLayout.NORTH);
        contentPane.add (new JScrollPane (textArea),
            BorderLayout.CENTER);
    }

    private void readFile () {
        DefaultEditorKit kit = new DefaultEditorKit ();
        try {
            kit.read (new FileReader ("Test.java"),
                textArea.getDocument (), 0);
        }
        catch (Exception ex) { ex.printStackTrace (); }
    }

    class ControlPanel extends JPanel {
        public ControlPanel () {
```

```

final JCheckBox cb = new JCheckBox ("wrap");
add (cb);
cb.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        if (cb.isSelected ())
            textArea.setLineWrap (true);
        else
            textArea.setLineWrap (false);
    }
});

```

13.4 JScrollBar

Swing 的滚动体系结构（由 JViewport 类和 JScrollPane 类组成）在大多数情况是够用了。如本章前面所说明的，Swing 滚动体系结构依靠的是一个固定尺寸的视口，这个视口显示组件的一部分，通常这个组件比与之相关联的视口大（至少在一维上比与之相关联的视口大）。

然而，很多情况下这种视口/视图滚动模式不能满足需要。例如，如果与视口相关联的视图包含大量的数据，那么由于性能和资源的考虑，建立视图和通过调整视口中视图的位置来进行滚动是不切实际的。幸运的是，Swing 提供了滚动条类，它通过实现手动滚动来替代 Swing 的滚动体系。

13.4.1 使用 Swing 的 JScrollBar 类进行手动滚动

有时构造一个能在视口中滚动的视图（又称作组件）是不切实际的。例如，让我们想一想这样一个小应用程序，它实现一个可滚动的列表视图，这个列表中有每个美国公民的名字和社会保险号。要创建包含将近 2 亿 5 千万个条目的要求就是不允许的，更别提所涉及的资源了。为了不创建包含所有名字和社会保险号的巨大的视图，最好的办法是当滚动列表时，就从存储设备中读取数据。这样的办法要求反转 Swing 的滚动体系结构，即不用固定大小的视口来滚动大的视图，而是当操纵与视图相关联的滚动条时，就重新绘制固定大小的视图。

图 13-15 所示的小应用程序实现了具有名字和社会保险号的一个滚动列表。这个小应用程序由一个面板和一个垂直滚动条组成，并说明了两件事，即 JScrollBar 实例的使用和前面所描述的反转的 Swing 滚动体系结构。为了简单的缘故，这个小应用程序只包含一个垂直滚动条，但维护垂直滚动条的技术也可以用在维护水平滚动条上。还是为了简单的缘故，图 13-15 所示的小应用程序不从磁盘读数据或实现数据缓存运算。然而，通过手动滚动一个固定大小的视图，这个小应用程序确实说明了反转的 Swing 滚动模型。

JScrollBar 是一个十分简单的组件，它显示箭头按钮和一个滑块（又称作拇指盖或滑杆）。JScrollBar 实例的模型是一个 BoundedRangeModel 接口的实现，它封装与最大值、最小值和可见部分值有关的一个 integer 值的操作（如最小值 \leq 这个整数值 \leq 这个整数值 + 可见部分值 \leq 最大值）。

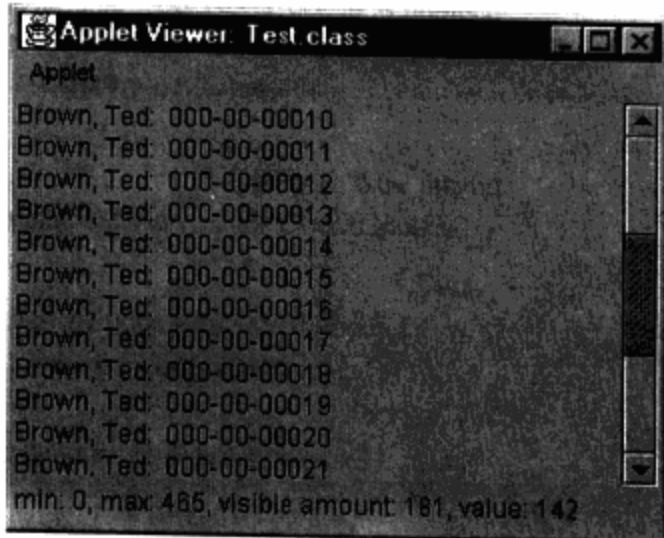


图 13-15 用滚动条手工滚动

图 13-15 所示的小应用程序为了显示名字和社会保险号实现了一个 JPanel 类的扩展。这个面板定义字符串数组（代表名字和社会保险号的固定集）。这个面板除跟踪显示这些字符串的字体的高度外，还跟踪在这个面板顶部显示的那个字符串的索引值。

SSPanel 类重载 paintComponent () 来绘制这些字符串。首先，调用 super. paintComponent () 来清除这个面板的背景。保存这个面板的前景色，然后在调用 super. paintComponent () 后再恢复这个面板的前景色，因为 JPanel. paintComponent () 把传送给它的图形的颜色设置为这个面板的背景色，有关 JPanel 类及它的不透明属性的更多信息，请参见 12.1 节 “JPanel”。

```
class SSPanel extends JPanel {
    private int topIndex = 0;
    private int fh;

    private String [] data = {
        "Brown, Ted: 000 - 00 - 0001", "Brown, Ted: 000 - 00 - 0002",
        "Brown, Ted: 000 - 00 - 0003", "Brown, Ted: 000 - 00 - 0004",
        "Brown, Ted: 000 - 00 - 0005", "Brown, Ted: 000 - 00 - 0006",
        "Brown, Ted: 000 - 00 - 0007", "Brown, Ted: 000 - 00 - 0008",
        "Brown, Ted: 000 - 00 - 0009", "Brown, Ted: 000 - 00 - 00010",
        "Brown, Ted: 000 - 00 - 00011", "Brown, Ted: 000 - 00 - 00012",
        "Brown, Ted: 000 - 00 - 00013", "Brown, Ted: 000 - 00 - 00014",
        "Brown, Ted: 000 - 00 - 00015", "Brown, Ted: 000 - 00 - 00016",
        "Brown, Ted: 000 - 00 - 00017", "Brown, Ted: 000 - 00 - 00018",
        "Brown, Ted: 000 - 00 - 00019", "Brown, Ted: 000 - 00 - 00020",
        "Brown, Ted: 000 - 00 - 00021", "Brown, Ted: 000 - 00 - 00022",
        "Brown, Ted: 000 - 00 - 00023", "Brown, Ted: 000 - 00 - 00024",
        "Brown, Ted: 000 - 00 - 00025", "Brown, Ted: 000 - 00 - 00026",
        "Brown, Ted: 000 - 00 - 00027", "Brown, Ted: 000 - 00 - 00028",
        "Brown, Ted: 000 - 00 - 00029", "Brown, Ted: 000 - 00 - 00030",
    };
}
```

paintComponent 方法绘制这个面板中的字符串，这些字符串从字符串 data [topIndex] 开始，以数组中最后一个字符串或面板中最后可见的字符串结束。

```
...
public void paintComponent (Graphics g) {
    Color color = g.getColor ();
    super.paintComponent (g);
    g.setColor (color);

    Dimension size = getSize ();
    Insets insets = getInsets ();
    int y = insets.top;

    for (int i = topIndex; i < data.length; ++i, y += fh) {
        g.drawString (data [i], 0, y);

        if (y + fh > size.height - insets.bottom)
            break;
    }
}
```

SSPanel 类实现一个方法，这个方法根据距面板顶部的偏移（以像素为单位）来设置它的 topIndex。重载 getPreferredSize 方法以便返回一个尺寸，这个尺寸是面板中显示的最宽字符串的宽度和显示所有字符串所需的总高度。

```

...
public void setTopIndexByPixelValue (int pixelValue) {
    topIndex = pixelValue / fh;
}

public Dimension getPreferredSize () {
    Dimension dim = new Dimension ();
    Graphics g = getGraphics ();
    try {
        FontMetrics fm = g.getFontMetrics ();
        dim.width = fm.stringWidth (data [data.length - 1]);
        dim.height = fm.getHeight () * (data.length + 1);
    }
    finally {
        g.dispose ();
    }
    return dim;
}
}

```

有关 `getPreferredSize` 方法，应该注意两件事。

第一，处理通过调用 `getGraphics()` 而获得的图形。对通过调用 `getGraphics()` 而获得图形而言，这是必要的。有关图形和它们的使用介绍，请参见《Java2 图形设计，卷 I: AWT》。

第二，面板的首选高度实际上是比面板所能显示的字符串再多一个字符串所需的高度。不能使面板底部显示的字符串被截去了一部分，因此，把面板的首选高度设置的稍微大一些。

这个小应用程序创建垂直滚动条和 `SSPanel` 类的一个实例。这个面板被指定作为这个小应用程序内容窗格的中心组件，而垂直滚动条是内容窗格的右边组件。

当修改滚动条的值时，`JScrollBar` 的实例激发调整事件。结果，添加到这个小应用程序滚动条中的调整监听器用当前的滚动条值调用 `SSPanel.setTopIndexByPixelValue()`，然后重新绘制这个面板。这个监听器还调用这个小应用程序的 `showScrollBarValues` 方法，该方法在这个小应用程序的状态区显示这个滚动条的值。

```

public class Test extends JApplet {
    private JScrollBar vsb = new JScrollBar (JScrollBar.VERTICAL);
    private SSPanel panel = new SSPanel ();

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.add (panel, BorderLayout.CENTER);
        contentPane.add (vsb, BorderLayout.EAST);

        vsb.addAdjustmentListener (new AdjustmentListener () {
            public void adjustmentValueChanged (
                AdjustmentEvent e) {
                JScrollBar sb = (JScrollBar) e.getSource ();
                showScrollBarValues ();

                panel.setTopIndexByPixelValue (e.getValue ());
                panel.repaint ();
            }
        });
    }
}

```

最后，当重新调整这个小应用程序的大小时，重载这个小应用程序的 `paint` 方法以确保更

新了这个滚动条。

这个面板的可见量或它的可见部分是它的当前高度，它的最大高度是这个面板的首选高度。如果这个面板的可见量比这个面板的最大高度大的话（即这个面板中的所有字符串都是可见的，不需要使用滚动条），则将滚动条的值设置为 0，此时，把滚动条的可见性设置为 false。

如果这个面板的可见量比其最大高度小（即不是所有的字符串都可以显示，因此，需要滚动条），则滚动条的值被设置为最小值（介于滚动条的当前值和最大值减可见量之间），而滚动条被设置为可见的。这导致滚动条的值保持不变，直到这个小应用程序的高度调整很大，以至于使在面板顶部显示的字符串发生了变化。

```
...
public void paint (Graphics g) {
    Dimension pref = panel.getPreferredSize ();
    Dimension size = panel.getSize ();
    int extent = size.height, max = pref.height;
    int value = Math.max (0,
        Math.min (vsb.getValue (), max - extent));

    vsb.setVisible (extent < max);
    vsb.setValues (value, extent, 0, max);

    showScrollBarValues ();
    super.paint (g);
}
```

例 13-11 列出了对图 13-15 所示的小应用程序做了一些修改后的代码，因此，这个小应用程序就不在这里列出来了。

Swing 提示

为获得最大性能而反转滚动模型

Swing 的滚动模型建立在固定大小的视口基础上，这种视口通常显示较大视图的一部分。

虽然 swing 滚动模型对大多数滚动应用是足够了，但是，为了显示有大量数据的视图，由于性能和资源的原因，可能这种视口/视图模型不能工作。在这种情况下，最好反转滚动模型，即提供一个固定大小的视图，当与之相关联的滚动条被操纵时，就重新绘制这个视图。

可以把许多策略应用到反转的滚动模型中，以便进一步增强性能。例如，缓存下一次和前一次的信息块、把图形从屏外缓存拷贝到视图的屏上代表中。

13.4.2 块增量和单元增量

JScrollBar 实例定义单元增量和块增量，通常单元增量和块增量分别定义滚动一项（又称单元）所需的像素数和在当前可见区域中显示的项数。例如，在文本编辑器中，单元增量代表一行文本，块增量代表一页文本。JScrollBar 实例的单元增量和块增量的初始值与界面样式有关。

除了使用由滚动条定义的单元增量和块增量来分别滚动一行和一页外，图 13-16 所示的小应用程序与图 13-15 所示的小应用程序是相同的。

在图 13-16 所示的小应用程序中，左上图显示了这个小应用程序开始时的样子。右上图显示在滚动条的向下箭头被激活一次后小应用程序的样子。激活滚动条的箭头按钮导致滚动条滚动一个单元增量。

左下图显示在滚动条的向上按钮被激活后的小应用程序，此时引起面板向上滚动一个单元。右下图显示当鼠标在滚动条的槽中按下后的小应用程序的样子，此时，滚动了一个块。

这个小应用程序滚动条的单元增量被定义为与这个面板的字体高度相同。因此，当激活滚动条的箭头按钮时，这个面板按一行的方式滚动。

滚动条的块增量被设置为面板的可见行数减去滚动条的单元增量。这种设置导致这个面板按可见行数减一的方式滚动，即面板底部的一行被滚动到面板的顶部。图 13-16 所示的小应用程序调用 JScrollBar 的 setUnitIncrement 和 setBlockIncrement 方法，如下所示：

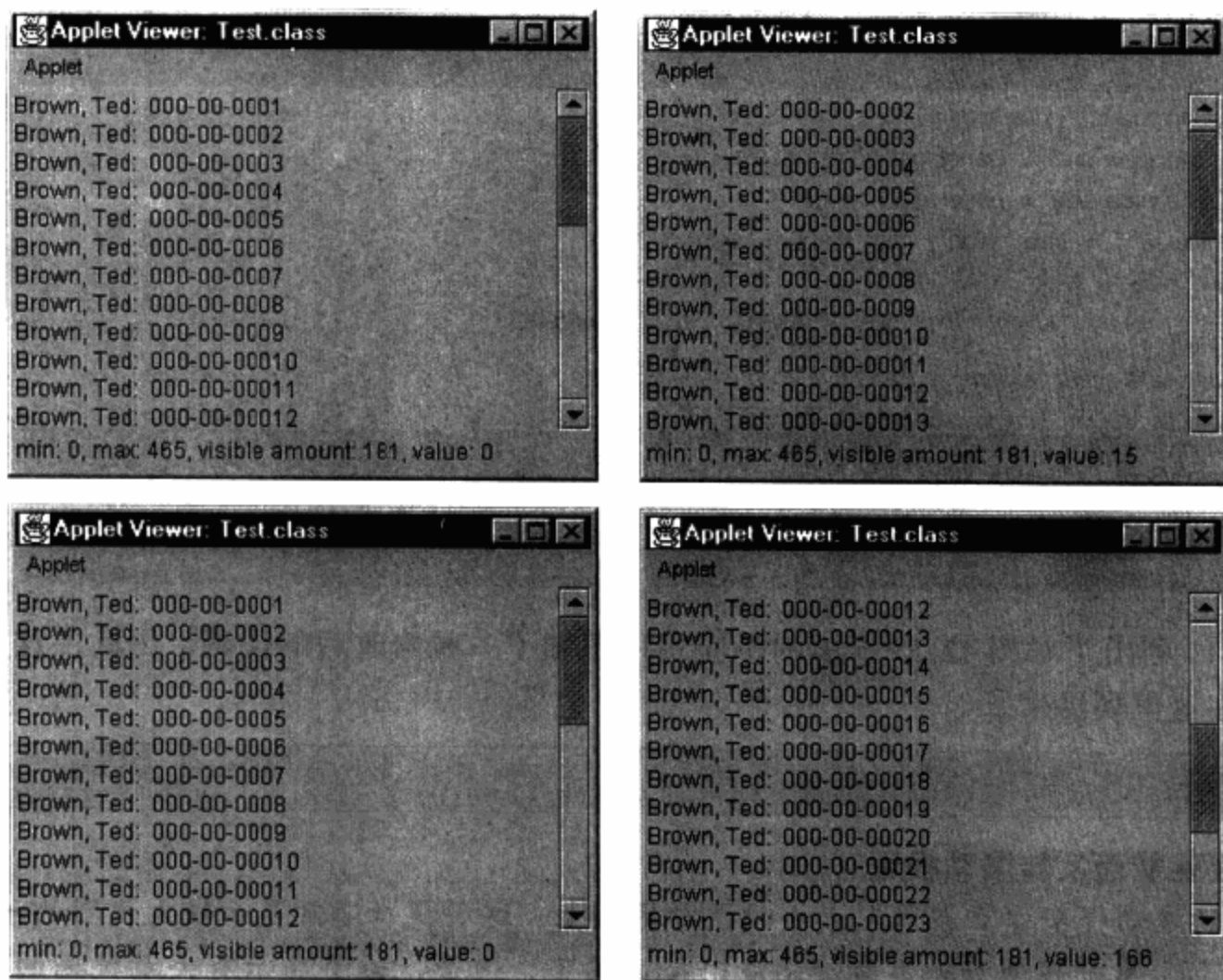


图 13-16 指定 JScrollBar 的块增量和单元增量

```
public class Test extends JApplet {
    ...
    public void paint (Graphics g) {
        Dimension pref = panel.getPreferredSize ();
        Dimension size = panel.getSize ();

        int extent = size.height, max = pref.height;
        int value = Math.max (0,
            Math.min (vsb.getValue (), max - extent));

        vsb.setVisible (extent < max);

        vsb.setUnitIncrement (panel.getUnitHeight ());
        vsb.setBlockIncrement (extent - vsb.getUnitIncrement ());

        vsb.setValues (value, extent, 0, max);

        showScrollBarValues ();
        super.paint (g);
    }
    ...
}
```



```

}
class SSPanel extends JPanel {
    private int topIndex = 0;
    private int fh;
    ...
    public Dimension getPreferredSize () {
        ...
        Graphics g = getGraphics ();
        ...
        try {
            FontMetrics fm = g.getFontMetrics ();
            fh = fm.getHeight ();
            ...
        }
        finally {
            g.dispose ();
        }
        ...
    }
    public int getUnitHeight () {
        return fh;
    }
}
...
}

```

因为图 13-16 所示的小应用程序除为滚动条设置了单元增量和块增量外，其他都于图13-15所示的小应用程序相同，所以这个小应用程序的剩余部分就不在这里讨论了。有关这个小应用程序的其余部分，请参见 13.4.1 节“用 Swing 的 JScrollBar 类进行手动滚动”。

例 13-11 列出了图 13-16 所示的小应用程序的完整代码。

例 13-11 为 JScrollBar 的实例指定单元增量和块增量

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.event. * ;

public class Test extends JApplet {
    private JScrollBar vsb = new JScrollBar (JScrollBar.VERTICAL);
    private SSPanel panel = new SSPanel ();

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.add (panel, BorderLayout.CENTER);
        contentPane.add (vsb, BorderLayout.EAST);
        vsb.addAdjustmentListener (new AdjustmentListener () {
            public void adjustmentValueChanged (
                AdjustmentEvent e) {
                JScrollBar sb = (JScrollBar) e.getSource ();
                showScrollBarValues ();
                panel.setTopIndexByPixelValue (e.getValue ());
                repaint ();
            }
        });
    }
}

```

```

    }
    public void paint (Graphics g) {
        Dimension pref = panel.getPreferredSize ();
        Dimension size = panel.getSize ();

        int extent = size.height, max = pref.height;
        int value = Math.max (0,
            Math.min (vsb.getValue (), max - extent));

        vsb.setVisible (extent < max);
        vsb.setUnitIncrement (panel.getUnitHeight ());
        vsb.setBlockIncrement (extent - vsb.getUnitIncrement ());
        vsb.setValues (value, extent, 0, max);
        showScrollBarValues ();
        super.paint (g);
    }

    private void showScrollBarValues () {
        showStatus ("min: " + vsb.getMinimum () +
            ", max: " + vsb.getMaximum () +
            ", visible amount: " +
            vsb.getVisibleAmount () +
            ", value: " + vsb.getValue ());
    }
}

class SSPanel extends JPanel {
    private int topIndex = 0;
    private int fh;

    private String [] data = {
        "Brown, Ted: 000 - 00 - 0001", "Brown, Ted: 000 - 00 - 0002",
        "Brown, Ted: 000 - 00 - 0003", "Brown, Ted: 000 - 00 - 0004",
        "Brown, Ted: 000 - 00 - 0005", "Brown, Ted: 000 - 00 - 0006",
        "Brown, Ted: 000 - 00 - 0007", "Brown, Ted: 000 - 00 - 0008",
        "Brown, Ted: 000 - 00 - 0009", "Brown, Ted: 000 - 00 - 00010",
        "Brown, Ted: 000 - 00 - 00011", "Brown, Ted: 000 - 00 - 00012",
        "Brown, Ted: 000 - 00 - 00013", "Brown, Ted: 000 - 00 - 00014",
        "Brown, Ted: 000 - 00 - 00015", "Brown, Ted: 000 - 00 - 00016",
        "Brown, Ted: 000 - 00 - 00017", "Brown, Ted: 000 - 00 - 00018",
        "Brown, Ted: 000 - 00 - 00019", "Brown, Ted: 000 - 00 - 00020",
        "Brown, Ted: 000 - 00 - 00021", "Brown, Ted: 000 - 00 - 00022",
        "Brown, Ted: 000 - 00 - 00023", "Brown, Ted: 000 - 00 - 00024",
        "Brown, Ted: 000 - 00 - 00025", "Brown, Ted: 000 - 00 - 00026",
        "Brown, Ted: 000 - 00 - 00027", "Brown, Ted: 000 - 00 - 00028",
        "Brown, Ted: 000 - 00 - 00029", "Brown, Ted: 000 - 00 - 00030",
    };

    public void paintComponent (Graphics g) {
        Color color = g.getColor ();
        super.paintComponent (g);
        g.setColor (color);

        Dimension size = getSize ();
        Insets insets = getInsets ();
        int y = insets.top;

        for (int i = topIndex; i < data.length; ++i, y += fh) {
            g.drawString (data [i], 0, y);
            if (y + fh > size.height - insets.bottom)

```

```

        break;
    }
    public void setTopIndexByPixelValue (int pixelValue) {
        topIndex = pixelValue / fh;
    }
    public int To () {
        return fh;
    }
    public Dimension getPreferredSize () {
        Dimension dim = new Dimension ();
        Graphics g = getGraphics ();
        try {
            FontMetrics fm = g.getFontMetrics ();
            fh = fm.getHeight ();

            dim.width = fm.stringWidth (data [data.length - 1]);
            dim.height = fm.getHeight () * (data.length + 1);
        }
        finally {
            g.dispose ();
        }
        return dim;
    }
}

```

组件总结 13-3 总结了 JScrollBar。

组件总结 13-3 JScrollBar

模型: BoundedRangeModel

UI 代表: javax.swing.plaf.basic.BasicScrollBarUI

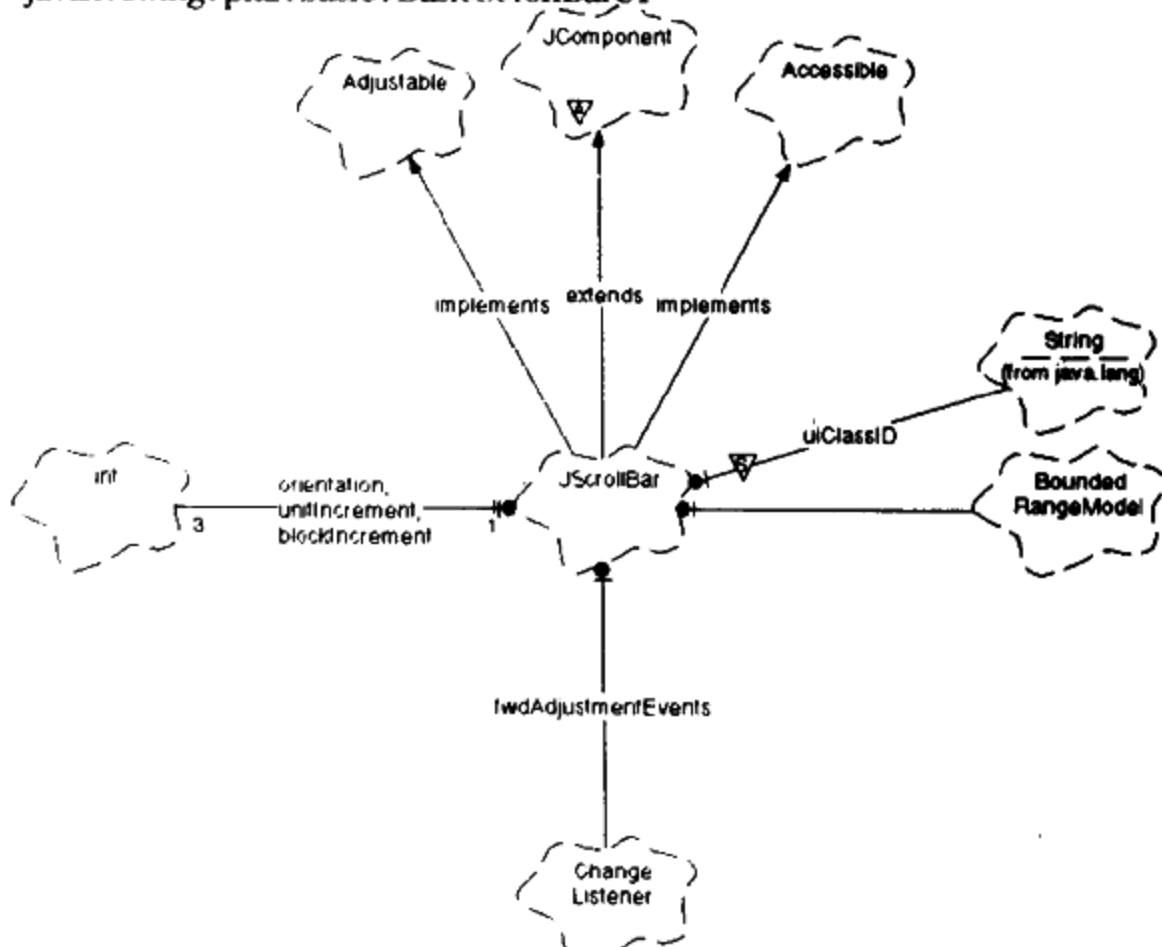


图 13-17 JScrollBar 类图

绘制器： ——
编辑器： ——
激发的事件： java.awt.event.AdjustmentEvent
替换： java.awt.ScrollBar
类图： 见图 13-17

JScrollBar 扩展 JComponent 并实现 Adjustable 接口和 Accessible 接口。实现 Adjustable 接口以便保留与 AWT 的 Scrollbar 的兼容。有关 JScrollBar 与 java.awt.Scrollbar 类兼容的更多信息，请参见 13.4.6 节“AWT 兼容”。

JScrollBar 维护一些 private integer 值，并维护对其模型和一个 ChangeListener 实例的 protected 引用。这个变化监听器监听由滚动条模型所激发的变化事件，这些事件被转换为调整事件，然后传送给已向滚动条登记的调整监听器。

13.4.3 JScrollBar 属性

表 13-4 列出了由 JScrollBar 类维护的属性。

表 13-4 JScrollBar 属性

属性名	数据类型	属性类型 ^②	访问 ^③	缺省值 ^④
blockIncrement	int	B	SG	L&F
maximum	int	A	CSG	100
minimum	int	A	CSG	0
model	BoundedRangeModel	B	SG	Default - Bounded - RangeModel
orientation	int	B	CSG	VERTICAL
unitIncrement	int	B	SG	L&F
value	int	A	CSG	0
valueIsAdjusting	boolean	——	SG	false
visibleAmount ^①	int	A	CSG	0

① 可见量的模型属性被称为可见部分
② B = 关联的（激发 PropertyChangeEvent）/C = 约束的/I = 索引的/
S = 简单的/Ch = 激发 ChangeEvent/A = 激发 AdjustmentEvent
③ C = 可在创建时设置/G = 获取方法/S = 设置方法
④ L&F = 与界面样式有关

blockIncrement——一个块请求所引起的滚动条值的改变量，通常这个值是一页数据。
maximum——滚动条所代表的最大值。
minimum——滚动条所代表的最小值。
model——滚动条的模型，它是 BoundedRangeModel 接口的一个实现。
orientation——滚动条的方向，是 JScrollBar.HORIZONTAL 或 JScrollBar.VERTICAL。
unitIncrement——一个单元请求所引起的滚动条值的改变量，通常它的值是一行数据。
value——由滚动条的滑块的左边界或上边界所代表的当前值。
valueIsAdjusting——如果为 true，则滚动条的滑块正在被拖动。
visibleAmount——当前可见的视图量。滚动条滑块的大小与可见量成比例。

注意 模型属性激发调整事件。与大多数其他的 Swing 组件不同，当修改模型属性时，其他的 Swing 组件将激发变化事件，而 JScrollBar 激发调整事件以响应对模型属性

的修改，以便维护与 AWT 的 Scrollbar 类的兼容性。

13.4.4 JScrollbar 事件

如前所述，当修改 JScrollbar 实例的模型属性（maximum、minimum、value 和 visibleAmount）时，JScrollbar 的实例激发调整事件。与 JScrollbar 的实例有关的所有其他属性（valueIsAdjusting 除外）是受约束的属性，即对这些属性的修改将激发属性变化事件。

虽然 JScrollbar 的实例在拖动滑块时会激发一个调整事件，但是，当滑块正在被拖动时，不总是需要滚动与容器相关联的滚动条的内容。因此，可以检测 valueIsAdjusting 属性以便确定滑块是否正在被拖动。

图 13-18 所示的小应用程序把一个调整监听器添加到它的滚动条中。如果滚动

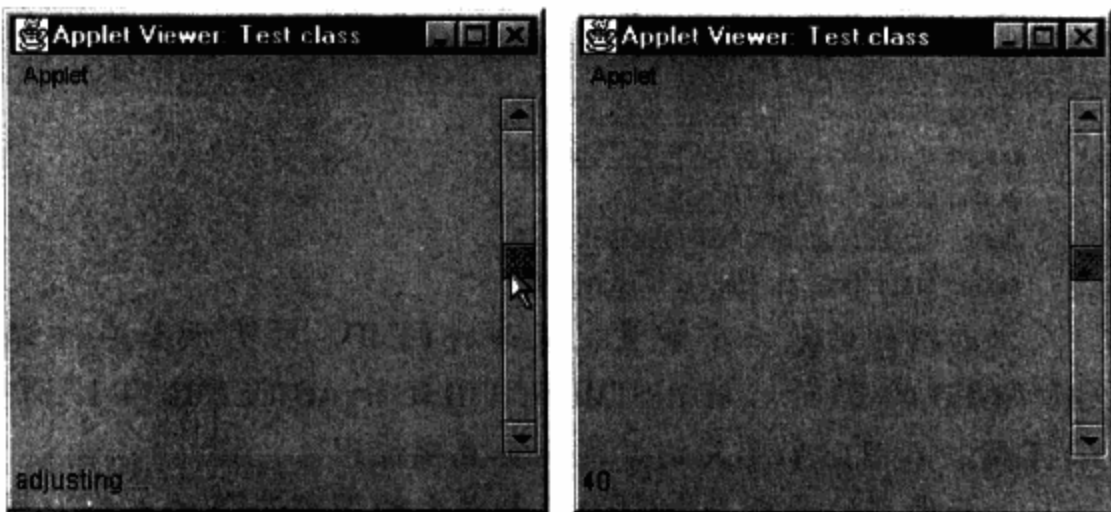


图 13-18 JScrollbar 的 valueIsAdjusting 属性

条的 valueIsAdjusting 属性是 true，则这个小应用程序在它的状态区显示字符串“adjusting...”，如左图所示。如果滚动条的 valueIsAdjusting 属性是 false，则这个小应用程序显示与滚动条相关联的值，如右图所示。

例 13-12 列出了图 13-18 所示的小应用程序的代码。

例 13-12 使用 JScrollbar 的 valueIsAdjusting 属性

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JScrollbar sb = new JScrollbar ();

        contentPane.add (sb, BorderLayout.EAST);

        sb.addAdjustmentListener (new AdjustmentListener () {
            public void adjustmentValueChanged (
                AdjustmentEvent e) {
                JScrollbar jsb = (JScrollbar) e.getAdjustable ();

                if (jsb.getValueIsAdjusting ())
                    showStatus ("adjusting ...");
                else
                    showStatus (Integer.toString (e.getValue ()));
            }
        });
    }
}
```

AdjustmentEvent 的这些实例以 `adjustmentValueChanged` 方法为参数，这个方法是由 `AdjustmentListener` 接口定义的唯一方法。类总结 13-3 总结了 AdjustmentEvent 类。

类总结 13-3 java.awt.AdjustmentEvent

扩展: `java.awt.AWTEvent`

1. 常量

```
public static final int ADJUSTMENT_FIRST
public static final int ADJUSTMENT_LAST
public static final int ADJUSTMENT_VALUE_CHANGED

public static final int BLOCK_DECREMENT
public static final int BLOCK_INCREMENT
public static final int TRACK
public static final int UNIT_DECREMENT
public static final int UNIT_INCREMENT
```

上面所列的前三个常量定义事件的 ID。所有调整事件都有一个 `ADJUSTMENT_VALUE_CHANGED` 的 ID 号。`ADJUSTMENT_FIRST` 和 `ADJUSTMENT_LAST` 定义由 AdjustmentEvent 类使用的 ID 范围，而且它们不能由开发人员直接使用。

上面所列的最后五个常量代表发生的调整事件的类型。遗憾的是，所有由 `JScrollBar` 实例激发的调整事件都是 `TRACK` 类型的，因为 `JScrollBar` 的实例创建调整事件。由于变化事件是无状态的，所以让 `JScrollBar` 的实例来确定事件的准确类型是不可能的。

2. 构造方法

```
public AdjustmentEvent (Adjustable source, int id, int type, int value)
```

AdjustmentEvent 是用事件源、事件 ID、事件类型和事件源的值作为参数来构造的。`JScrollBar` 参与了 AdjustmentEvent 的创建，而 `JScrollBar` 的用户通常不参与创建 AdjustmentEvent。

3. 方法

```
public Adjustable getAdjustable ()
public int getAdjustmentType ()
public int getValue ()
public String paramString ()
```

`getAdjustable` 方法返回对激发事件的 `JScrollBar` 的一个引用。如上所述，`getAdjustmentType` 总是返回 `AdjustmentEvent.TRACK` 的一个值。`getValue` 方法返回滚动条的当前值。`paramString` 方法把事件的类型转换为字符串，但是这个方法总是返回“TRACK”，因为由 `JScrollBar` 的实例激发的调整事件只有这一种类型。

Swing 提示

由 JScrollBar 的实例激发的调整事件

与大多数其他的 Swing 组件激发变化事件不同，`JScrollBar` 类激发调整事件以便维护与 AWT 滚动条组件最大程度的兼容。

`JScrollBar` 的实例将被通知，当用一个变化事件（由滚动条的模型激发）来修改它们的模型属性时。当 `JScrollBar` 的一个实例从它的模型那里接收到一个变化事件时，它就创建一个调整事件，并把这个调整事件发送给它的调整监听器。

因为从由滚动条模型激发的变化事件中获得的唯一可使用的信息是事件源，所以调整事件不包含任何状态信息。因此，由 `JScrollBar` 的实例激发的所有调整事件都有一个 `Adjust-`

mentEvent.TRACK 的事件 ID。

13.4.5 JScrollBar 类总结

类总结 13-4 列出了 JScrollBar 的 public 和 protected 变量和方法。

类总结 13-4 JScrollBar

1. 构造方法

```
public JScrollBar ()
public JScrollBar (int)
public JScrollBar (int orientation, int value, int extent, int minimum, int maximum)
```

无参数的构造方法以最小值为 0、最大值为 100、值 (value) 和可见部分值 (可见量) 均为 0 为参数来创建 JScrollBar 的一个实例。

上面所列的第二个构造方法以指定滚动条方向的 integer 值为参数。这个参数的有效值是 SwingConstants.HORIZONTAL 或 SwingConstants.VERTICAL。

上面所列的第三个构造方法所带的参数是指示滚动条方向的 integer 值、值 (value)、可见部分值、最小值和最大值。

2. 方法

(1) 调整事件

```
public void addAdjustmentListener (AdjustmentListener)
protected void fireAdjustmentValueChanged (int id, int type, int value)
public void removeAdjustmentListener (AdjustmentListener)
```

上面所列的这些方法把调整监听器添加到 JScrollBar 的实例中，或从 JScrollBar 的实例中删除调整监听器。用 fireAdjustmentValueChanged 方法把调整事件发送给调整监听器。这个方法总是带入相同的事件 ID 参数 (ADJUSTMENT_VALUE_CHANGED) 和相同的事件类型参数 (TRACK)。JScrollBar 的扩展可以重载这个方法，以便在需要时以不同的方式来激发事件。

(2) 属性访问方法

```
public int getBlockIncrement ()
public int getBlockIncrement (int)
public int getMaximum ()
public Dimension getMaximumSize ()
public int getMinimum ()
public Dimension getMinimumSize ()
public BoundedRangeModel getModel ()
public int getOrientation ()
public int getUnitIncrement ()
public int getUnitIncrement (int)
public int getValue ()
public boolean getValuesAdjusting ()
public int getVisibleAmount ()

public void setBlockIncrement (int)
public void setEnabled (boolean)
public void setMaximum (int)
public void setMinimum (int)
public void setModel (BoundedRangeModel)
public void setOrientation (int)
public void setUnitIncrement (int)
```



```
public void setValue (int)
public void setValuesAdjusting (boolean)
public void setValue (int value, int extent, int min, int max)
public void setVisibleAmount (int)
```

上面所列的是 JScrollBar 属性的访问方法。getBlockIncrement (int) 和 getUnitIncrement (int) 方法带入一个指定滚动方向的 integer 值。这个值被 JScrollBar 方法忽略，这两个方法分别返回块增量或单元增量。JScrollBar 扩展可以重载这些方法，以便根据滚动方向来返回不同的增量。

(3) 可访问性和插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public ScrollBarUI getUI ()
public String getUIClassID ()
public void updateUI ()
```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

13.4.6 AWT 兼容

JScrollBar 百分之百地与 java.awt.Scrollbar 兼容，从表 13-5 可以看得很清楚。JScrollBar 不仅实现与 java.awt.Scrollbar 完全相同的方法，而且它还提供相同的构造方法。

JScrollBar 与 java.awt.Scrollbar 之间的唯一的 API 差别是：由 java.awt.Scrollbar 实现的两个获取增量的方法没有由 JScrollBar 实现。java.awt.Scrollbar 的 getLineIncrement 方法和 getPageIncrement 方法分别被替换为 getUnitIncrement () 方法和 getBlockIncrement () 方法。

表 13-5 java.awt.Scrollbar 方法和 JScrollBar 的对应方法

java.awt.Scrollbar 方法	JScrollBar 的对应方法
void addAdjustmentListener (AdjustmentListener)	void addAdjustmentListener (AdjustmentListener)
int getBlockIncrement ()	int getBlockIncrement ()
int getMaximum ()	int getMaximum ()
int getMinimum ()	int getMinimum ()
int getOrientation ()	int getOrientation ()
int getUnitIncrement ()	int getUnitIncrement ()
int getValue ()	int getValue ()
int getVisibleAmount ()	int getVisibleAmount ()
void removeAdjustmentListener (Adjustment - Listener)	void removeAdjustmentListener (Adjustment - Listener)
void setBlockIncrement (int)	void setBlockIncrement (int)
void setMaximum (int)	void setMaximum (int)
void setMinimum (int)	void setMinimum (int)
void setOrientation (int)	void setOrientation (int)
void setUnitIncrement (int)	void setUnitIncrement (int)
void setValue (int)	void setValue (int)
void setValues (int, int, int, int)	void setValues (int, int, int, int)
void setVisibleAmount (int)	void setVisibleAmount (int)

13.5 本章回顾

Swing 的滚动体系结构说明了自初始的 AWT 以来，Java 的用户界面组件有了很大的改进。

起初, AWT 甚至没有提供一个滚动窗格组件。在早期的 AWT 中, 滚动一个组件意味着要手动放置一个带滚动条的容器, 处理滚动条事件的底层代码并滚动组件。

随着 AWT 1.1 的出现, 出现了滚动窗格容器。然而, AWT 滚动窗格不太强壮。AWT 不提供单独的视口类, AWT 滚动窗格中的视口是一个逻辑构造, 所以不能重复使用内置在滚动窗格类中的视口功能。而且, 虽然 AWT 滚动窗格支持的滚动条显示策略与 Swing 滚动窗格类支持的滚动条显示策略相同, 但是, 不能把不同的策略用到不同的滚动条上。最后, AWT 滚动窗格不支持头部、角部或透明, 而 Swing 的 JScrollBar 则支持这些功能。

Swing 的滚动体系结构是 Java GUI 开发人员工具包中受欢迎的附件。随着出现了单独的视口类, 开发人员能够重复使用和扩展视口的功能。要了解扩展 JViewport 来支持拖动和自动滚动视口的视图的样例, 请参见 4.6 节“自动滚动”。另外, 为了得到可拖动的头部等特性, 可以把行头部和列头部添加到一个视口中, 图 13-9 所示的小应用程序说明了可拖动的头部特性。

最后, Swing 提供了滚动条 (以 JScrollBar 类的形式), 手动滚动时需要这种滚动条。通过用 JScrollBar 的实例来实现直接滚动, 可以反转 Swing 的滚动体系结构, 使性能和资源得到最大限度的利用。

第 14 章 窗口和对话框

Swing 的窗口 (window)、窗体 (frame) 和对话框 (dialog) 是分别扩展 AWT 的 window 类、Frame 类和 Dialog 类的重量组件。当这三个组件都是窗口时，这三个组件之间的差别是不明显的，因此，有时在给定情况下要确定使用哪个组件是很困难的。为了澄清这些差别，表 14-1 列出了与这三个组件有关的一些属性。

表 14-1 窗口、窗体和对话框属性^{①,②}

属性	窗口	窗体	对话框
模态	否	否	否/CSG
可调整大小	否	是/SG	是/SG
标题栏	否	是	是
边框	否	是	是
标题	否	是/CSG	是/CSG
菜单栏	否	是/SG	否
焦点管理器	是	是	是
警告字符串	是/G	是/G	是/G
图标图像 ^③	否	是/SG	否
链接到一个窗体	是	否	是

① 是/否指缺省的属性状态。

② C = 在构造时可设置，S = 可使用的设置方法，G = 可使用的获取方法（即 get... () 或 is... ()）

③ 不是所有的平台都支持窗口的图标化。

窗口是这三个组件中最基本的组件，事实上，java.awt.Window 是 Frame 和 Dialog 的超类。窗口没有边框、标题栏或菜单栏，而且不能调整其大小。如果需要在其他组件之上的无边框矩形区域中显示某些内容，则窗口是最合适的。

窗体是 Window 的一个扩展，它有边框、标题栏并可以调整大小。当所需的应用程序的窗口需要图标化或要带菜单栏的话，则应该选择使用窗体。

对话框也是 Window 的一个扩展，与窗体一样，它也有边框、标题栏并可以调整大小。对话框可以是模态的，而窗体和窗口则不能是模态的。当需要使用一个临时窗口来捕获用户输入时，则应该选择使用对话框。

Swing 的窗口、窗体和对话框把一个根窗格添加到它们各自扩展的 AWT 类中。结果，通过把组件添加到在根窗格中的内容窗格中，这些组件就被添加到 Swing 的窗口、窗体和对话框中了。另外，通过设置内容窗格的布局管理器而不是容器本身的布局管理器来设置布局管理器。

根窗格（由 JRootPane 类表示）和窗体（由 JFrame 类表示）不在本章介绍。根窗格和窗体在“Swing 的基础知识”一章中介绍。AWT 的 Frame 类、Window 类和 Dialog 类在《Java2 图形设计，卷 I: AWT》做了详细的介绍。

14.1 JWindow

JWindow 是一个重量 Swing 组件，它扩展 java.awt.Window 并把 JRootPane 的一个实例作为窗口唯一的组件来进行安装。JWindow 的实例没有边框或菜单栏，而且不能调整窗口的大小。通

常，如果要在所有其他组件之上显示的无边框区域中显示组件或图形，则使用 Swing 的窗口。例如，用 JWindows 的实例来实现 Swing 的工具提示。

图 14-1 所示的应用程序图解说明了 JWindow 类的另一种使用——一个 splash 屏幕。这个应用程序创建一个窗口，它包含了一个配备了图像图标 of JLabel 实例。这个窗口一直显示，直到在这个窗口内检测到一个鼠标按下事件。

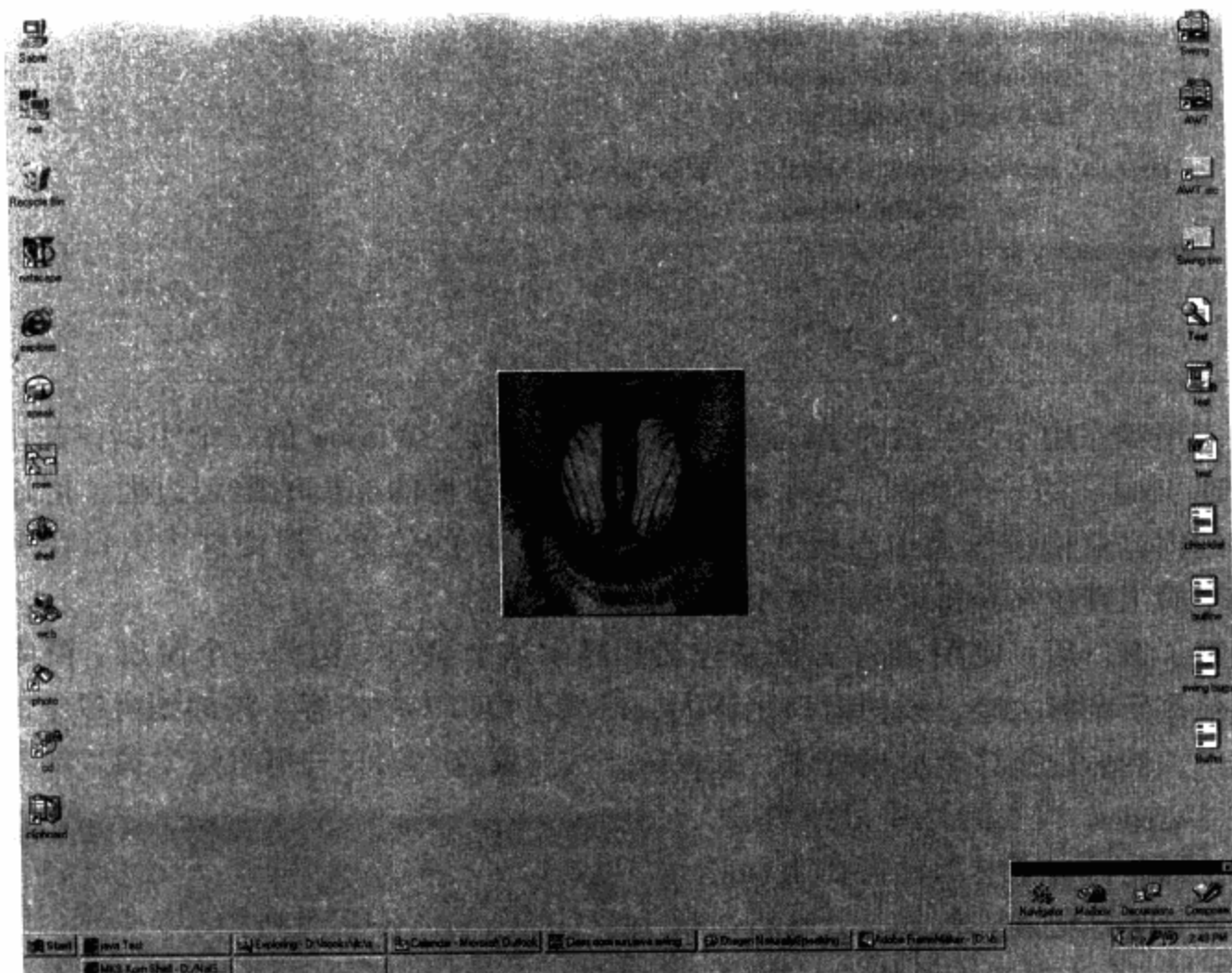


图 14-1 一个作为 splash 窗口使用的 JWindow 实例

例 14-1 列出了图 14-1 所示的小应用程序的代码。

例 14-1 使用 JWindow 来实现一个 splash 窗口

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JFrame {
    Toolkit toolkit = Toolkit.getDefaultToolkit();
    JWindow window = new JWindow();
    JLabel label = new JLabel(new ImageIcon("mandrill.jpg"));

    static public void main(String[] args) {
        JFrame frame = new Test();
    }

    public Test() {
        label.setBorder(BorderFactory.createRaisedBevelBorder());
        window.getContentPane().add(label, BorderLayout.CENTER);
        centerWindow();
        window.show();

        window.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
            }
        });
    }
}
```

```

        window.dispose ();
        System.exit (0);
    }
}

private void centerWindow () {
    Dimension scrnSize = toolkit.getScreenSize ();
    Dimension labelSize = label.getPreferredSize ();
    int      labelWidth = labelSize.width,
           labelHeight = labelSize.height;

    window.setLocation (scrnSize.width/2 - (labelWidth/2),
                        scrnSize.height/2 - (labelHeight/2));
    window.pack ();
}

```

这个应用程序用不带参数的 JWindow 构造方法来创建 JWindow 的一个实例，并用一个图像图标来创建 JLabel 的一个实例。这个标签有从边框库获得的突出的雕刻边框，把这个标签添加到窗口的内容窗格中作为中心组件。接着，计算窗口的位置，以便这个窗口在屏幕上居中（使用 AWT 的工具包来获得屏幕的大小）。

在配置了窗口后，则用 show 方法来包装和显示这个窗口。包装一个窗口、窗体或对话框导致它们被重新调整大小，以便窗口足够大，能放下窗口中的内容。添加到窗口中的鼠标监听器清理窗口，然后退出这个应用程序。dispose 方法隐藏这个窗口并清理这个窗口的本地资源。

记住 JWindow 的实例是重量组件是很重要的，因此，它不继承任何在 JComponent 类中实现的功能。例如，让 JWindow 的实例有一个边框是不可能的，因为 JWindow 类不实现一个 setBorder 方法。

然而，JWindow 的实例包含一个轻量组件（以 JRootPane 的一个实例的形式），而且可以像操纵任何轻量组件那样操纵根窗格。例如，图 14-2 所示的小应用程序创建 JWindow 的一个实例，这个实例为其根窗格设置一个突出的雕刻边框。因为根窗格被指定为这个窗口的居中组件，所以它完全填充这个窗口，因此，

为根窗格设置一个边框就有效地为这个窗口设置了一个边框。而且，在这个窗口的根窗格上还添加了一个菜单栏。

使用如图 14-2 示出的窗口要十分小心。与窗体不同，Swing 的窗口没有最小化、最大化或关闭框控制。通常，如果需要的是一个应用程序窗口，则最好使用 JFrame 的一个实例。图 14-2 示出的小应用程序及例 14-2 所列的代码主要是想说明如何操纵窗口的根窗格。

例 14-2 一个作为应用程序窗口使用的 JWindow 实例

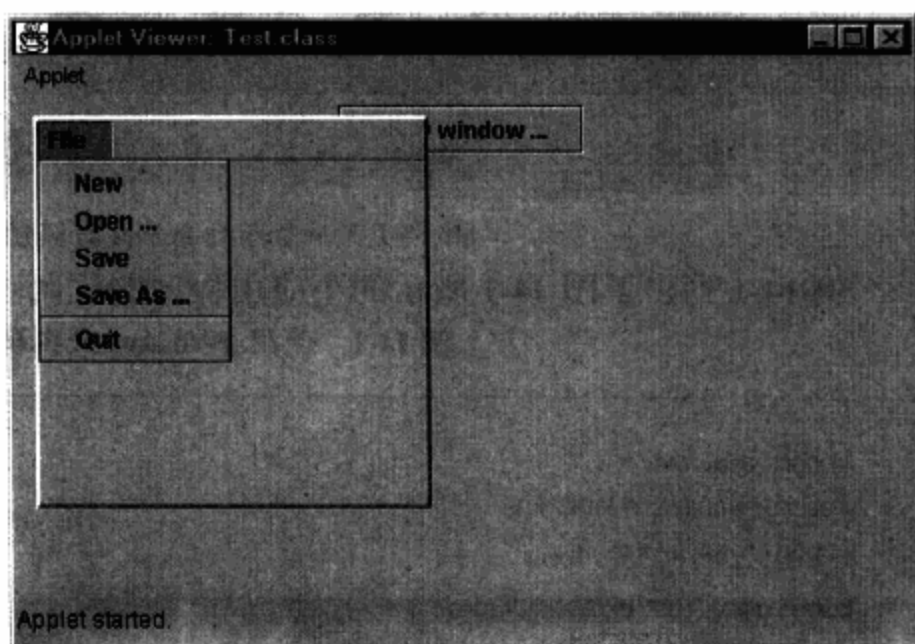


图 14-2 一个作为应用程序窗口使用的 JWindow 实例

```
import java.awt.*;
```

```

import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    JWindow window = new JWindow ();
    JMenuBar menuBar = new JMenuBar ();
    JMenu fileMenu = new JMenu ("File");
    JMenuItem quitItem;

    public Test () {
        final Container contentPane = getContentPane ();
        JButton button = new JButton ("show window ...");
        JRootPane windowRootPane = window.getRootPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        fileMenu.add ("New");
        fileMenu.add ("Open ...");
        fileMenu.add ("Save");
        fileMenu.add ("Save As ...");
        fileMenu.addSeparator ();
        fileMenu.add (quitItem = new JMenuItem ("Quit"));

        menuBar.add (fileMenu);

        windowRootPane.setMenuBar (menuBar);
        windowRootPane.setBorder (
            BorderFactory.createRaisedBevelBorder ());

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                Point pt = contentPane.getLocation ();

                SwingUtilities.convertPointToScreen (
                    pt, contentPane);

                //display 10 pixels below and to the right of the
                // upper-left corner of the content pane

                window.setBounds (pt.x + 10, pt.y + 10, 200, 200);
                window.show ();

                quitItem.addActionListener (new ActionListener () {
                    public void actionPerformed (ActionEvent e) {
                        window.dispose ();
                    }
                });
            }
        });
    }
}

```

这个小应用程序用 JWindow 无参数构造方法来创建 JWindow 的一个实例，并获取对窗口根窗格的一个引用。

这个小应用程序还创建一个用于显示这个窗口的按钮。创建一个菜单栏和一个菜单，菜单被添加到菜单栏中，且菜单栏被添加到窗口的根窗格中。从边框库获得一个边框并把它作为窗口的根窗格的边框。

当激活这个小应用程序中的按钮时，就设置这个窗口的范围，以便这个窗口在这个小应用

程序的左上角附近显示，这个窗口的宽是 200 个像素，高是 200 个像素。

因为 JWindow 的实例没有关闭框，所以，与 JFrame 的实例一样，添加一个监听器到清理窗口的 Quit 菜单项中。

组件总结 14-1 总结了 JWindow 组件。

组件总结 14-1 JWindow

- 模型： ——
- UI 代表： ——
- 绘制器： ——
- 编辑器： ——
- 激发的事件： ——
- 替换： java.awt.Window
- 类图：

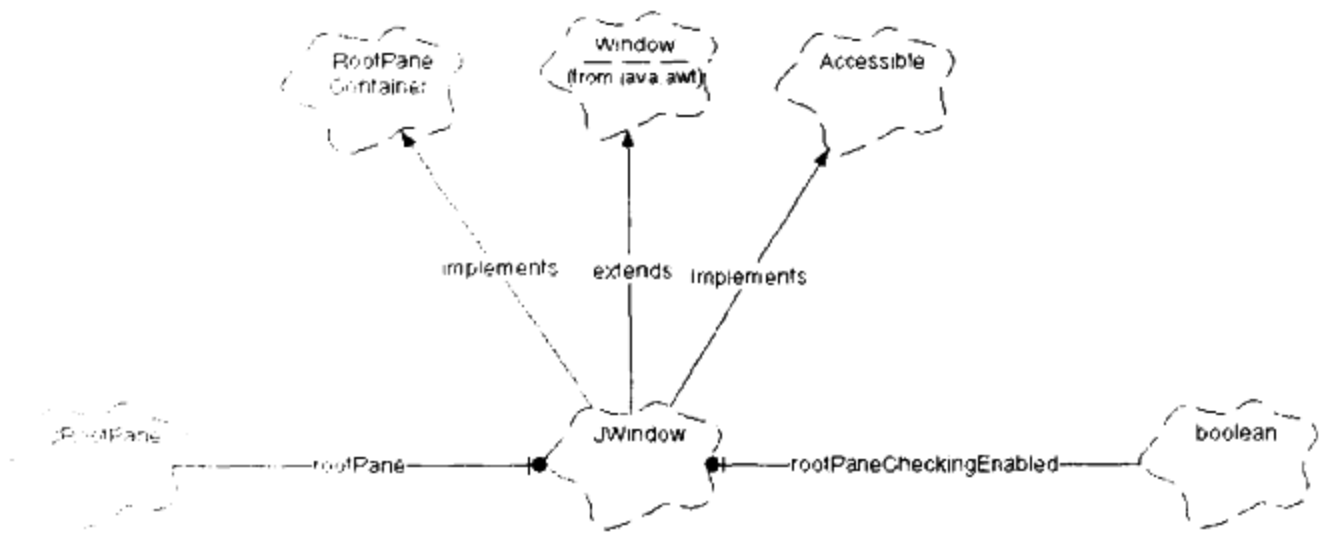


图 14-3 JWindow 类图

JWindow 是一个重量容器，所以没有模型、UI 代表、绘制器和编辑器。

JWindow 类扩展 java.awt.Window 并实现 Accessible 接口和 RootPaneContainer 接口。JWindow 维护两个 protected 引用，一个是对根窗格的引用，另一个是对 boolean 变量的引用，这个变量跟踪是否启用了根窗格检查。有关根窗格检查的更多信息，请参见 12.2 节“JRootPane”。

14.1.1 JWindow 属性

表 14-2 列出了由 JWindow 类维护的属性。

表 14-2 JWindow 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
contentPane	Container	S	SG	JPanel 实例
glassPane	Component	S	SG	Component 实例
layeredPane	JLayeredPane	S	SG	JLayeredPane 实例
rootPane	JRootPane	S	SG	JRootPane 实例

① B = 关联的（激发 PropertyChangeEvent）/C = 受约束的/I = 索引的/
S = 简单的/Ch = 激发 ChangeEvent
② C = 可在创建时设置/G = 获取方法/S = 设置方法
③ I&F = 与界面样式有关

contentPane—— 一个容器，JWindow 的一个实例中的组件就放在这个容器中。

glassPane—— 在根窗格中的所有其他组件上面“悬浮”的组件。

layeredPane—— 每个根窗格都包含一个 JLayeredPane 实例，该实例包含根窗格的菜单栏和内容窗格。

rootPane—— 直接包含在 JWindow 的实例中的唯一组件。

第 12 章“轻量容器”详细介绍了根窗格和它们所包含的组件（玻璃窗格、层窗格和内容窗格）。

14.1.2 JWindow 类总结

类总结 14-1 列出了 JWindow 的 public 和 protected 变量和方法。

类总结 14-1 JWindow

1. 构造方法

```
public JWindow ()
public JWindow (Frame)
```

必须把所有的重量窗口都链接到一个窗体中，这个窗体被称作这些窗口的拥有者。因此，JWindow 类提供了一个以 java.awt.Frame 引用为参数的构造方法。

JWindow 类还提供一个无参数的构造方法，它把窗口链接到一个共享的、不可见的窗体中，这个窗体是所有用无参数构造方法创建的 JDialog 和 JWindow 实例的拥有者。这个共享窗体可以用 SwingUtilities.getSharedOwnerFrame 方法来获得。

2. 方法

(1) 根窗格/添加组件/设置布局管理器

```
protected void addImpl (Component, Object, int)
protected JRootPane createRootPane ()
protected boolean isRootPaneCheckingEnabled ()
protected void setRootPaneCheckingEnabled (boolean)
protected void setRootPane (JRootPane)

public void setLayout (LayoutManager)
protected void windowInit ()
```

上面所列的第一组的五个方法用于设置窗口的根窗格。与其他实现 RootPaneContainer 接口的 Swing 组件一样，JWindow 不允许把组件直接添加到窗口中，也不允许为窗口设置布局管理器。然而，JWindow 本身必须添加根窗格并设置它的布局管理器。为满足这种要求，把 setRootPaneCheckingEnabled 和 isRootPaneCheckingEnabled 方法用于允许 JWindow 的实例们添加它们的根窗格和设置它们的布局管理器，同时，不允许外界做这些事情。

createRootPane 方法创建 JRootPane 的一个实例。这个方法是 protected 方法，以便在需要时 JWindow 的扩展可以插入一个指定的根窗格。

setLayout 方法会弹出一个异常信息，指出只能用窗口本身来设置布局管理器。

通过 JWindow 的构造方法来调用 windowInit 方法。这个方法唯一的功能是设置根窗格并在设置根窗格后启用根窗格检查。这个方法是 protected 方法，以便 JWindow 扩展在构造时可以执行一些其他的任务或在需要时重载缺省的行为。

(2) RootPaneContainer 方法

```
public Container getContentPane ()
public Component getGlassPane ()
```

```

public JLayeredPane getLayeredPane ()
public JRootPane getLayeredPane ()
public void setContentPane (Container)
public void setGlassPane (Component)
public void setLayeredPane (JLayeredPane)

```

上面所列的这些方法由 `RootPaneContainer` 接口来定义。`RootPaneContainer` 接口和 `JRootPane` 类在第 12 章“轻量容器”中介绍。

(3) 可访问的相关内容

```
public AccessibleContext getAccessibleContext ()
```

`JWindow` 实现 `Accessible` 接口，因此，实现上述的方法。

14.1.3 AWT 兼容

`JWindow` 是 `java.awt.Window` 的一个扩展，因此继承了由 AWT 的 `Window` 类实现的所有 `public` 方法。

Swing 窗口与 AWT 窗口最主要的差别是 Swing 窗口包含一个根窗格。结果，通过访问窗口的内容窗格来把组件添加到 Swing 窗口中。另外，不能为 Swing 窗口直接设置布局管理器，而是必须为窗口的内容窗格设置布局管理器。

14.2 JDialog

与 `JWindow` 一样，`JDialog` 是一个重量 Swing 容器，它包含 `JRootPane` 的一个实例，这是它唯一的组件。与 Swing 窗口不同，Swing 对话框有一个边框和一个标题栏。通常，Swing 对话框在它们的标题栏上还有一个关闭框，用这个关闭框来清除这个对话框。对话框标题栏中组件的样子与窗口系统有关。

Swing 对话框可以是模态的，即只要对话框正在显示，就不能访问这个对话框的父窗口中的其他窗口。而且，显示模态对话框的线程被阻塞，直到清除这个对话框。Swing 对话框在缺省时，不是模态的。

`JDialog` 的实例是基础，本质上它们是配备了一个 `JRootPane` 实例的本地对话框。直接使用 `JDialog` 类来创建对话框涉及布局这个对话框所包含的各个组件、创建清除对话框的按钮、和安装响应激活的按钮的监听器。Swing 提供一个类 (`JOptionPane` 类)，它自动完成在创建和显示对话框时所需要做的操作，在 14.3 节“`JOptionPane`”中对这个类进行了介绍。

图 14-4 所示的小应用程序包含一个按钮，激活它就会显示一个对话框。图 14-4 中的左图显示这个小应用程序开始时的样子，右图显示已经显示对话框后这个小应用程序的样子。

这个小应用程序创建 `ConstraintsPanel` 的一个实例，这个实例包含除 OK 按钮、Apply 按钮和 Cancel 按钮以外对话框中的所有组件。`ConstraintsPanel` 的实现与这里讨论的概念无关，因此在这里就不介绍了。在本书后面的 CD 盘上有 `ConstraintsPanel` 的实现。

这个小应用程序创建 `JPanel` 的一个实例，这个实例用来包含 OK 按钮、Apply 按钮和 Cancel 按钮。这个小应用程序还创建 `JDialog` 的一个实例，指定 `null` 作为这个对话框的父窗体、`Constraints Dialog` 作为这个对话框的标题、模式属性为 `true`。

通过调用这个对话框的 `getContentPane ()` 来获得对这个对话框内容窗格的一个引用，并且把约束 (`constraints`) 面板和按钮面板添加到内容窗格中，分别把它们作为中心组件和下边组件。然后调用 `pack` 方法来包装这个对话框，`pack` 方法调整这个对话框的大小，以便它能够容

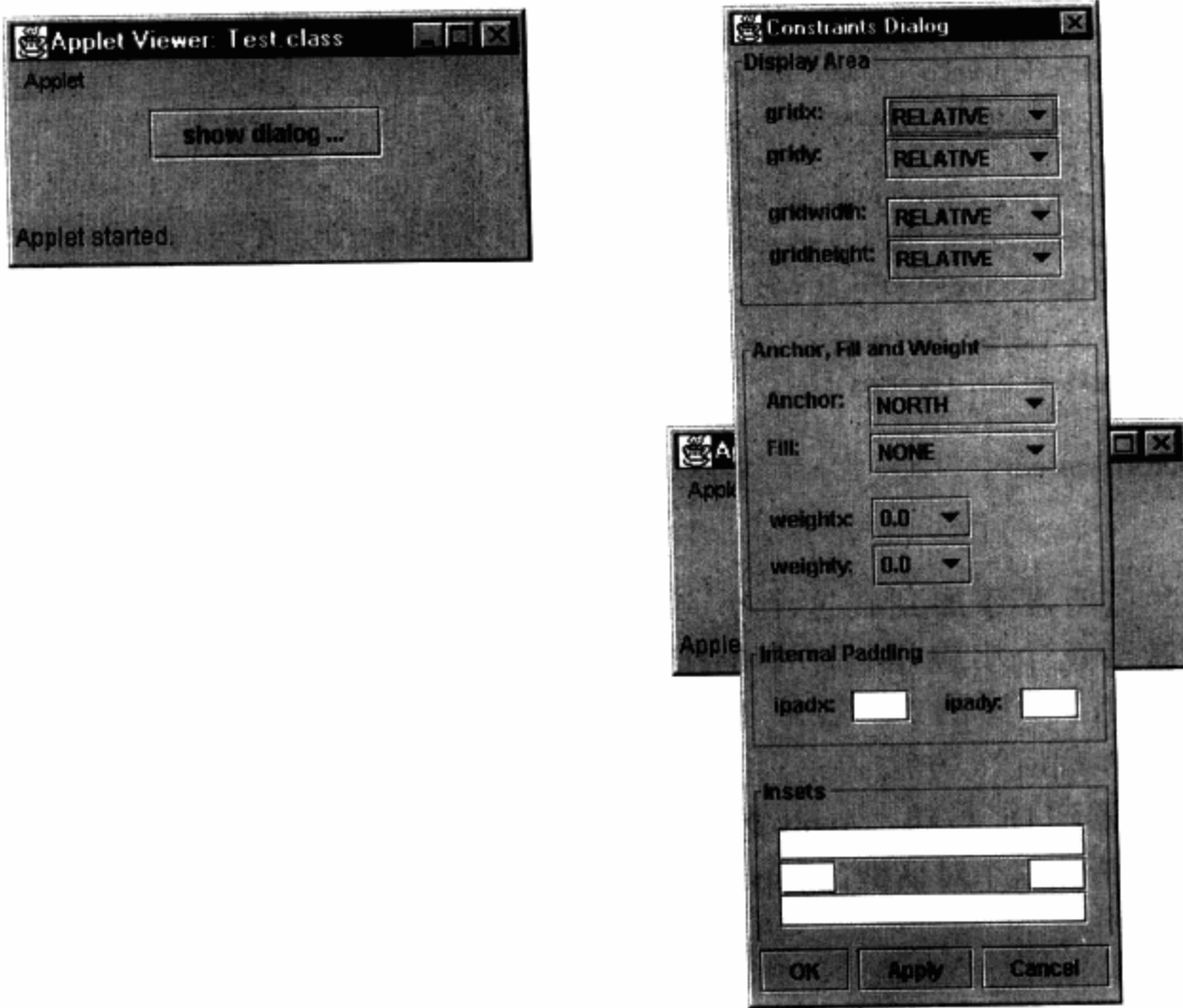


图 14-4 运行中的 JDialog

纳下它的组件。

注意 pack 方法中的一个错误是给出的对话框大小比它应该具有的大小要稍微短一些。

```
public class Test extends JApplet {
    private ConstraintsPanel cp = new ConstraintsPanel ();
    private JPanel buttonsPanel = new JPanel ();

    private JButton showButton = new JButton ("show dialog ..."),
        okButton = new JButton ("OK"),
        applyButton = new JButton ("Apply"),
        cancelButton = new JButton ("Cancel");

    private JButton [] buttons = new JButton [] {
        okButton, applyButton, cancelButton,
    };

    private JDialog dialog = new JDialog (null, // owner
        "Constraints Dialog", // title
        true); // modal

    public Test () {
        Container contentPane = getContentPane ();
        Container dialogContentPane = dialog.getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (showButton);

        dialogContentPane.add (cp, BorderLayout.CENTER);
        dialogContentPane.add (buttonsPanel, BorderLayout.SOUTH);
        dialog.pack ();

        // setLocationRelativeTo must be called after pack ()
        // because dialog placement is based on dialog size.
    }
}
```

```
// Because the applet is not yet showing, calling
// setLocationRelativeTo () here causes the dialog to be
// shown centered on the screen.
//
// If setLocationRelativeTo () is not invoked, the dialog
// will be located at (0, 0) in screen coordinates.
//dialog.setLocationRelativeTo (this);

for (int i=0; i < buttons.length; ++i) {
    buttonsPanel.add (buttons [i]);
}
addButtonListeners ();
}
```

把一些监听器添加到在这个小应用程序中显示的按钮和在这个对话框中显示的按钮中。当激活这个小应用程序中的按钮时，它设置对话框的位置（该位置是相对于这个小应用程序本身的），然后显示这个对话框。在类总结 14-2 中介绍了 `JDialog.setLocationRelativeTo` 方法。

添加监听器到 OK 按钮、Apply 按钮和 Cancel 按钮中。激活 OK 按钮和 Cancel 按钮就会调用 `dispose` 方法来消除这个对话框，`dispose` 方法隐藏这个对话框并清除与这个对话框的窗口有关的本地资源。

```
...
private void addButtonListeners () {
    showButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            // calling setLocationRelativeTo () here causes
            // the dialog to be centered over the applet.

            dialog.setLocationRelativeTo (Test.this);
            dialog.show ();
        }
    });

    okButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            showStatus ("OK button Activated");
            dialog.dispose ();
        }
    });

    applyButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            showStatus ("Apply button Activated");
        }
    });

    cancelButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            showStatus ("Cancel button Activated");
            dialog.dispose ();
        }
    });
}
```

例 14-3 列出了图 14-4 所示的小应用程序的代码。

例 14-3 运行中的 JDialog

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private ConstraintsPanel cp = new ConstraintsPanel ();
    private JPanel buttonsPanel = new JPanel ();
    private JButton showButton = new JButton ("show dialog ..."),
        okButton = new JButton ("OK"),
        applyButton = new JButton ("Apply"),
        cancelButton = new JButton ("Cancel");

    private JButton [] buttons = new JButton [] {
        okButton, applyButton, cancelButton,
    };

    private JDialog dialog = new JDialog (null, // owner
        "Constraints Dialog", // title
        true); // modal

    public Test () {
        Container contentPane = getContentPane ();
        Container dialogContentPane = dialog.getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (showButton);

        dialogContentPane.add (cp, BorderLayout.CENTER);
        dialogContentPane.add (buttonsPanel, BorderLayout.SOUTH);
        dialog.pack ();

        // setLocationRelativeTo must be called after pack (),
        // because dialog placement is based on dialog size.
        // Because the applet is not yet showing, calling
        // setLocationRelativeTo () here causes the dialog to be
        // shown centered on the screen.
        //
        // If setLocationRelativeTo () is not invoked, the dialog
        // will be located at (0, 0) in screen coordinates.
        //dialog.setLocationRelativeTo (this);

        for (int i = 0; i < buttons.length; ++i) {
            buttonsPanel.add (buttons [i]);
        }
        addButtonListeners ();
    }

    private void addButtonListeners () {
        showButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                // calling setLocationRelativeTo () here causes
                // the dialog to be centered over the applet.
                dialog.setLocationRelativeTo (Test.this);
                dialog.show ();
            }
        });

        okButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {

```

```

        showStatus ("OK button Activated");
        dialog.dispose ();
    }
}
applyButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        showStatus ("Apply button Activated");
    }
});
cancelButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        showStatus ("Cancel button Activated");
        dialog.dispose ();
    }
});
}
}
}

```

组件总结 14-2 总结了 JDialog 组件。

组件总结 14-2 JDialog

模型: _____
 UI 代表: _____
 绘制器: _____
 编辑器: _____
 激发的事件: _____
 替换: java.awt.Dialog
 类图:

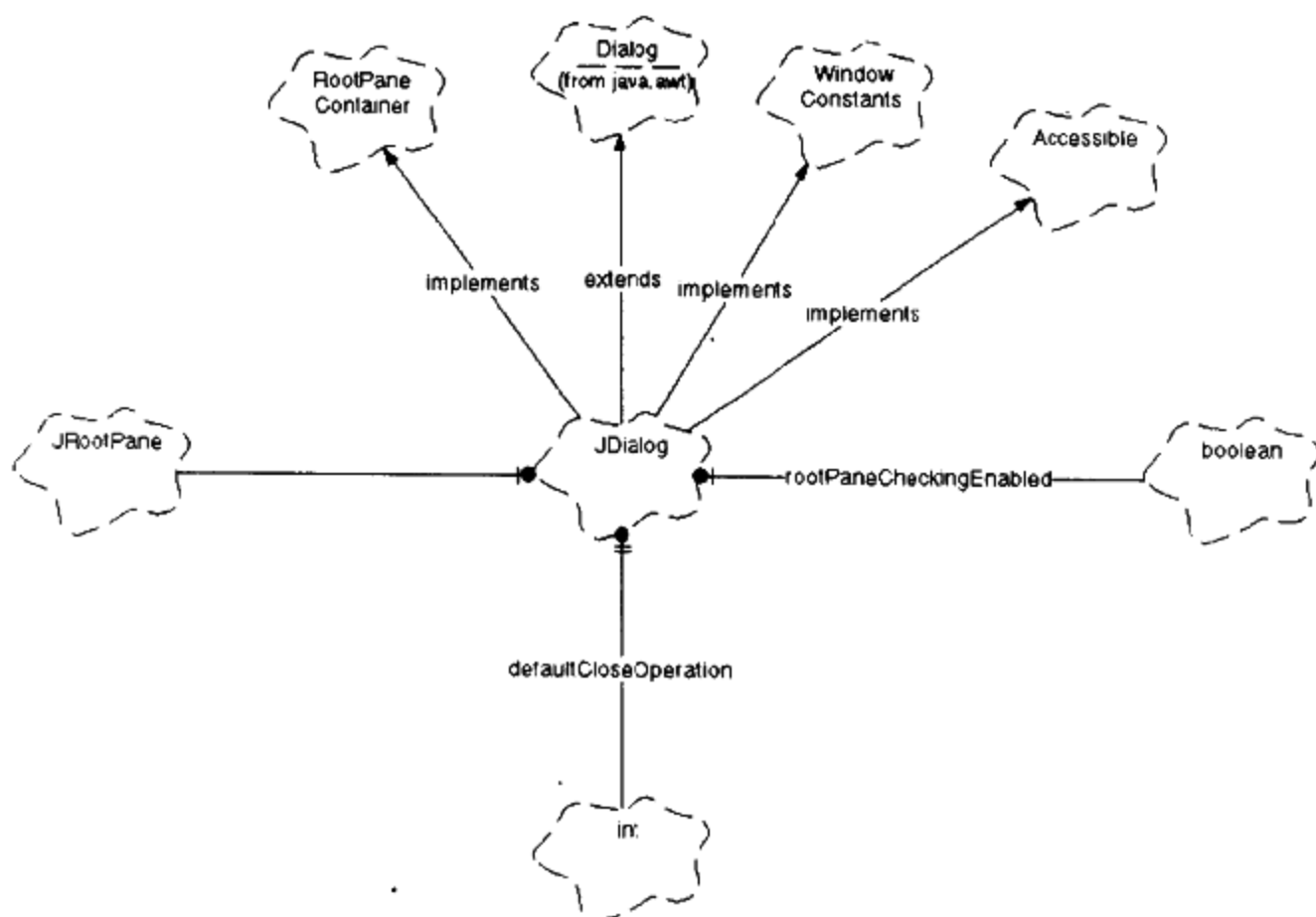


图 14-5 JDialog 类图

JDialog 类扩展 java.awt.Dialog 并实现 Accessible 接口、WindowConstants 接口和 RootPaneContainer 接口。与 JWindow 一样，JDialog 没有模型、UI 代表、绘制器或编辑器。

JDialog 类维护对它的根窗格的一些 protected 引用并维护一个 boolean 值，该值跟踪根窗格检查是否是允许的。JDialog 的每个实例还维护一个 protected integer 值，这个值指定缺省的关闭操作。有关缺省关闭操作的详细内容，请参见“JDialog 属性”。

14.2.1 JDialog 属性

表 14-3 列出了由 JDialog 类维护的属性。

表 14-3 JDialog 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
contentPane	Container	S	SG	JPanel 实例
defaultCloseOperation	int	S	SG	HIDE_ON_CLOSE
glassPane	Component	S	SG	组件实例
jMenuBar	JMenuBar	S	SG	null
layeredPane	JLayeredPane	S	SG	JLayeredPane 实例
location	Component	S	G	
relativeTo	JRootPane	S	SG	JRootPane 实例

① B = 关联的 (激发 PropertyChangeEvent) / C = 受约束的 / I = 索引的 /
S = 简单的 / Ch = 激发 ChangeEvent
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ I&F = 与界面样式有关

contentPane——一个容器，JDialog 实例中的组件被放置在这个容器中。

defaultCloseOperation——在关闭对话框时将要执行的一个操作。defaultCloseOperation 属性必须指定为下列 integer 值中的一个值：

- WindowConstants.HIDE_ON_CLOSE
- WindowConstants.DISPOSE_ON_CLOSE
- WindowConstants.DO_NOTHING_ON_CLOSE

glassPane——一个组件，它“悬浮”在根窗格中的所有其他组件之上。

jMenuBar——与一个对话框相关联的菜单栏。

layeredPane——每个根窗格都包含 JlayeredPane 的一个实例，这个实例包含根窗格的菜单栏和内容窗格。

locationRelativeTo——指定对话框将要显示的位置。如果传送给 setLocationRelativeTo () 的组件是 null，则对话框将在屏幕坐标的 (0, 0) 处显示。如果组件不是 null，但是在调用 setRelativeTo () 时是不可见的，则这个对话框将在屏幕中间显示。

如果组件不是 null 并且当调用 setRelativeTo () 时是可见的，则这个对话框将在这个指定的组件上居中显示。

rootPane——直接包含在 JDialog 的实例中的唯一组件。

14.2.2 JDialog 类总结

类总结 14-3 列出了 JDialog 的 public 和 protected 变量和方法。

类总结 14-2 JDialog**1. 构造方法**

```

public JDialog ()
public JDialog (Frame owner)
public JDialog (Frame owner, boolean modal)
public JDialog (Frame owner, String title)
public JDialog (Frame owner, String title, boolean modal)

```

JDialog 类提供上面所列的五个构造方法。无参数构造方法创建一个没有标题的非模态对话框，这个对话框以一个共享窗体作为它的拥有者。

上面所列的最后四个构造方法以作为对话框父窗体的窗体为参数。boolean 参数指定这个对话框是否是模态的，字符串参数代表在对话框标题栏中显示的标题。

2. 方法**(1) 根窗格/添加组件/窗口事件**

```

protected void addImpl (Component, Object, int)
protected JRootPane createRootPane ()
protected boolean isRootPaneCheckingEnabled ()
protected void setRootPaneCheckingEnabled (boolean)
protected void setRootPane (JRootPane)

protected void dialogInit ()
protected void processWindowEvent (WindowEvent)

```

上面所列的第一组 protected 方法用来设置对话框的根窗格。这些方法与 JWindow 类的同名方法相同。

dialogInit 方法由 JDialog 构造方法调用，而且它设置对话框的根窗格并允许窗口事件。

从 java.awt.Component 类重载 processWindowEvent 方法，以便进行将在下面介绍的对话框的缺省关闭操作。

(2) 缺省关闭操作/菜单栏/布局管理器

```

public int getDefaultCloseOperation ()
public void setDefaultCloseOperation (int)
public JMenuBar getJMenuBar ()
public void setJMenuBar (JMenuBar)

public void setLayout (LayoutManager)

```

与 JFrame 类一样，JDialog 维护一个缺省的关闭操作，这个缺省操作由 WindowConstants 接口定义的常数之一来定义。

与 AWT 对话框不同，Swing 对话框可以支持一个菜单栏，从上面所列的菜单栏访问方法中可以清楚地看到。

与 JFrame 一样，当允许根窗格检查时，如果试图设置对话框的布局管理器，则 JDialog 将重载 setLayout 来弹出一个错误信息。

(3) RootPaneContainer 方法

```

public Container getContentPane ()
public Component getGlassPane ()
public JLayeredPane getLayeredPane ()
public JRootPane getRootPane ()

public void setContentPane (Container)
public void setGlassPane (Component)

```

```
public void setLayeredPane (JLayeredPane)
```

上面所列的这些方法由 `RootPaneContainer` 接口定义。`RootPaneContainer` 接口和 `JRootPane` 类在“`JRootPane`”中介绍。

(4) 相对位置/更新

```
public void setLocationRelativeTo (Component)
```

```
public void update (Graphics)
```

可以用 `setLocationRelativeTo` 方法来设置一个对话框相对于一个组件的位置。有关 `setLocationRelativeTo` 方法的更多信息，请参见 14.2.1 “`JDialog` 属性”。

与大多数没有 UI 代表的 Swing 组件一样，`JDialog` 重载它的 `update` 方法来直接调用 `paint()`。这样做，就不会清除对话框的背景，因为缺省时，`update()` 清除背景，然后调用 `paint()`。

(5) 可访问的相关内容

```
public AccessibleContext getAccessibleContext ()
```

`JDialog` 实现 `Accessible` 接口，因此，实现上面所列的这个方法。有关 `Accessible` 接口和可访问性的更多信息，请参见 4.11 节“支持可访问性”。

14.2.3 AWT 兼容

`JDialog` 是 `java.awt.Dialog` 的一个扩展，因此继承了所有由 AWT 的 `Dialog` 类实现的 `public` 方法。Swing 对话框与 AWT 对话框最主要的区别是 Swing 对话框包含一个根窗格。所以，组件通过对话框的根窗格被添加到 Swing 的对话框中。另外，不能为 Swing 对话框直接设置布局管理器，而是必须为对话框的内容窗格设置布局管理器。

14.3 JOptionPane

选项窗格（由 `JOptionPane` 类表示）是打算放在对话框中的组件。选项窗格可以显示一个图标、一幅图像、一个或多个可选值和一行按钮。

选项窗格相当灵活，几乎可以在任何类型的对话框中使用。例如，可以用一个对 `object` 的引用来指定在选项窗格中显示的“消息”，即 `object` 是作为消息显示在选项窗格上的对象。消息对象显示的方式与对象的实际类型有关，字符串和组件都按原样显示，而图标被包装在 `JLabel` 的一个实例中。那些不是字符串、组件或图标的消息对象显示从这些对象的 `toString` 方法中返回的字符串。消息对象还可以是一个 `object` 数组，数组中的每一项都用上面所描述的方式显示，并按存储到数组中的顺序，从上到下地垂直摆放数组的项。

可以由选项窗格的界面样式来自动配置在一个选项窗格中显示的图标和按钮，或可以显式地指定这些图标和按钮的位置。`JOptionPane`

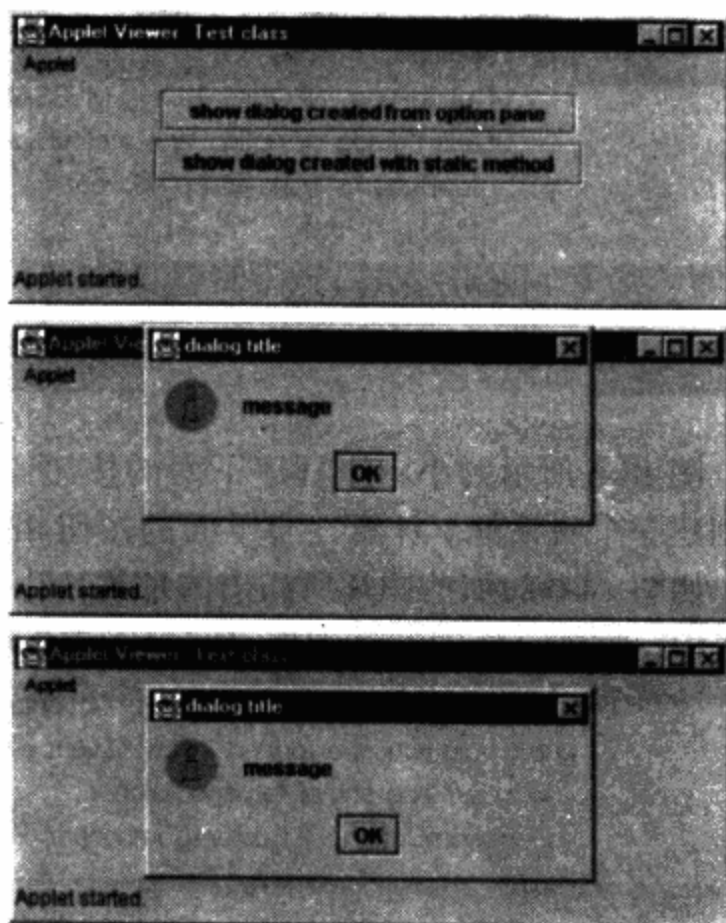


图 14-6 用 `JOptionPane` 来创建对话框

类提供两个属性（即 `icon` 和 `optionType`），这两个属性分别决定在选项窗格中使用的图标和按钮。另外，`JOptionPane` 类还提供了另一种属性（`options`），可以用这个属性来显式地指定这些按钮。

`JOptionPane` 类提供了一个简便的方法，这个方法创建一个对话框并在这个对话框中安装 `JOptionPane` 的一个实例，另外，`JOptionPane` 还提供多个 `static` 的简便方法，它们创建包含完全配置的选项窗格的对话框。

图 14-6 所示的小应用程序包含两个按钮，当这些按钮被激活时，将显示对话框。激活上面的按钮将创建 `JOptionPane` 的一个实例，接着创建一个包含选项窗格的对话框。激活下面的按钮将调用 `static JOptionPane.showMessageDialog` 方法来创建一个对话框。

图 14-6 中上面的图片显示这个小应用程序开始时的样子。中间的图片显示当激活上面的按钮时所创建的对话框，下面的图片显示当激活下面的按钮时所创建的对话框。

这个小应用程序给每个按钮都添加了一个动作监听器。上面按钮的监听器通过指定消息和 `JOptionPane.INFORMATION_MESSAGE` 的消息类型来创建一个 `JOptionPane` 实例。接着，调用 `JOptionPane.createDialog()` 来创建一个对话框，`createDialog` 方法的参数是一个父组件和对话框的标题。通过调用对话框的 `show` 方法来使这个对话框可见。

传送给 `JOptionPane.createDialog()` 的父组件定位这个对话框，这个对话框在父组件上居中显示。在这里，指定父组件为这样的一个按钮，激活这个按钮将创建这个对话框。

```
public class Test extends JApplet {
    ...
    private String title = "dialog title";
    private String message = "message";
    public Test () {
        ...
        topButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JOptionPane optionPane = new JOptionPane (
                    message, // message
                    JOptionPane.INFORMATION_MESSAGE); // messageType
                JDialog dialog = optionPane.createDialog (
                    topButton, // parentComponent
                    title); // title
                dialog.show ();
            }
        });
        ...
    }
}
```

图 14-6 所示的小应用程序下面的按钮监听器调用 `static JOptionPane.showMessageDialog` 方法来创建一个对话框。注意，与下面的按钮相关联的对话框的父组件还是显示这个对话框的按钮。如图 14-6 的那些图片所指出的那样，由这个小应用程序显示的这些对话框在这两个按钮上居中，这两个按钮被分别指定作为这两个对话框的父组件。

```
bottomButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        JOptionPane.showMessageDialog (
            bottomButton, // parentComponent
            message, // message
            title, // title
            JOptionPane.INFORMATION_MESSAGE); // messageType
    }
});
```

```

    }
    ...
    ...
}

```

由于这个小应用程序显示的这两个对话框都是模态对话框，即当显示对话框时，就不能访问这个对话框父组件中的窗口。另外，模态对话框还阻止显示这个对话框的线程的执行。所有由 `JOptionPane` 类创建的对话框都是模态的。

例 14-4 列出了图 14-6 所示小应用程序的完整代码。

例 14-4 用 `JOptionPane` 创建对话框

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
    private JButton topButton = new JButton (
        "show dialog created from option pane");
    private JButton bottomButton = new JButton (
        "show dialog created with static method");

    private String title = "dialog title";
    private String message = "message";

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (topButton);
        contentPane.add (bottomButton);

        topButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JOptionPane optionPane = new JOptionPane (
                    message, // message
                    JOptionPane.INFORMATION_MESSAGE); // messageType

                JDialog dialog = optionPane.createDialog (
                    topButton, // parentComponent
                    title); // title

                dialog.show ();
            }
        });

        bottomButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JOptionPane.showMessageDialog (
                    bottomButton, // parentComponent
                    message, // message
                    title, // title
                    JOptionPane.INFORMATION_MESSAGE); // messageType
            }
        });
    }
}

```

14.3.1 内部窗体

JOptionPane 类除提供了 static 方法（这些方法创建配备了选项窗格的对话框）外，JOptionPane 类还提供了创建内部窗体的方法。

图 14-7 所示的小应用程序包含一个按钮，激活这个按钮将创建一个内部窗体。

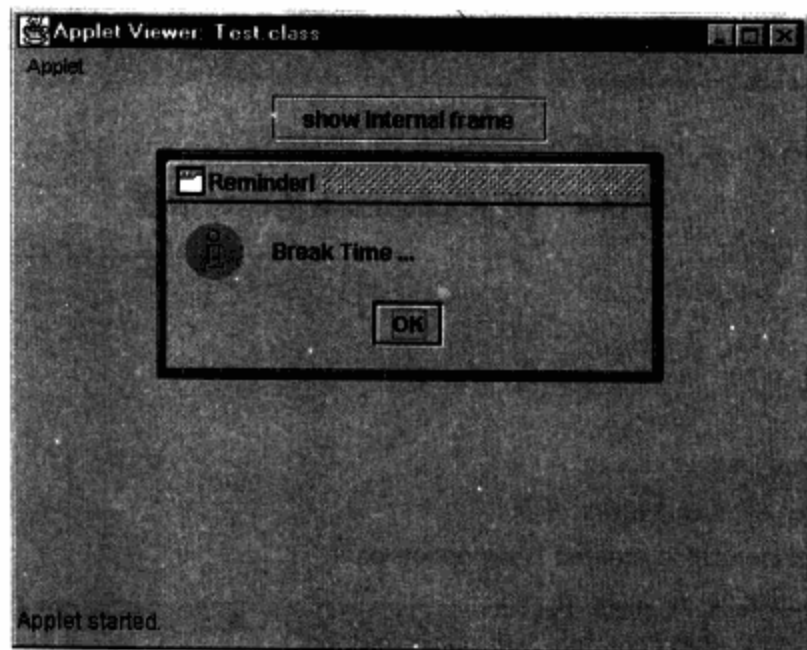


图 14-7 用 JOptionPane 创建内部窗体

例 14-5 列出了图 14-7 所示的小应用程序的代码。

例 14-5 用 JOptionPane 创建内部窗体

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JButton button = new JButton ("show internal frame");

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JOptionPane.showMessageDialog (
                    button, // parentComponent
                    "Break Time ...", // message
                    "Reminder!", // title
                    JOptionPane.INFORMATION_MESSAGE); // messageType
            }
        });
    }
}
```

这个小应用程序把一个动作监听器添加到它的按钮中，这个监听器与图 14-6 所示的小应用程序的监听器几乎完全相同。这两个监听器之间的差别是图 14-7 所示的小应用程序按钮的

监听器调用 `showInternalMessageDialog` 方法而不是调用 `showMessageDialog`。

Swing 提示

由 `JOptionPane` 类创建的对话框是模态对话框

由静态 `JOptionPane` 方法或 `JOptionPane.createDialog()` 创建的对话框是模态的。模态对话框禁止访问对话框父组件中的窗口，并且阻止显示对话框线程的执行。

因为由静态 `JOptionPane` 方法创建的对话框是模态的，所以调用 `JOptionPane.showXXXDialog` 后面的一行代码直到清除这个对话框后才会执行。

```
...
JOptionPane.showMessageDialog (parentComponent, "a message");

// the following line of code is not executed until the message
dialog is dismissed

someObject.someMethod ();
...
```

14.3.2 用 `JOptionPane` 静态方法创建对话框

`JOptionPane` 类提供 11 个构造方法，它们允许用各种配置来创建选项窗格。有关创建各种配置的 `JOptionPane` 的实例的小应用程序举例，请参见图 14-18。

`JOptionPane` 还提供一组 `static` 方法，这些方法创建包含选项窗格的对话框，这些选项窗格代表常用的对话框配置。表 14-4 总结了由 `JOptionPane static` 方法创建的对话框的类型。

表 14-4 由静态 `JOptionPane` 方法创建的对话框类型

对话框类型	描述	返回值	配置项
消息	显示一个消息和一个 OK 按钮	—	标题、消息、消息类型、图标
确认	询问一个问题，选择按钮来回答	代表所选按钮的整数	标题、消息、消息类型、图标选项类型 (按钮)
输入	用一个文本域、组合框或列表来提示输入 ^①	代表响应的字符串	标题、消息、消息类型、图标选择值、初始值
选项	除按钮行中的对象是完全可配置的外，其余与确认对话框相似	代表所选按钮的整数	标题、消息、消息类型、选项类型 (按钮)、图标选项、初始值

① 所使用的实际组件与界面样式有关

消息对话框是可以用 `JOptionPane static` 方法创建的最简单的对话框类型。消息对话框显示一个消息并有一个标签为 OK 的按钮。

确认对话框提出一个问题，通过选取在这个对话框中显示的按钮来回答这个问题。创建确认对话框的静态 `JOptionPane` 方法返回一个 `integer` 值，它指示被激活的那个按钮。在确认对话框中显示的按钮可以用预先确定的按钮集中的按钮来配置。表 14-5 显示了一个按钮配置列表，这个配置列表可以用 `optionType` 参数来指定。

输入对话框通过显示一个文本域、组合框或列表等组件来捕获输入。所使用的实际组件与界面样式有关，并且与为输入对话框指定的选取值有关。

选项对话框是唯一由静态 `JOptionPane` 方法创建的对话框类型，它允许完全配置在对话框的按钮行中的组件。

传送给 static JOptionPane 方法（创建表 14-4 中列出的对话框类型）的参数遵循一种一致的模式，因此表 14-5 中列出的对不同参数类型的描述是值的一看的。

表 14-5 用于构造 JOptionPane 对话框的参数

参数	描述	应用到
parent Component (Component)	这个对话框的父组件被设置为包含这个组件的窗体。通常，这个对话框在这个组件上居中显示，然而，对话框的位置是与界面样式有关的。	所有的对话框
message (Object)	在选项窗格中显示的消息。这个对象的数据类型决定消息是如何被显示的： Object []：数组中的每个对象都以下面概要的方式进行解释，并按顺序由上到下垂直排放。 String：字符串按原样显示。 Component：组件按原样显示。 Icon：图标被包装在一个 JLabel 中。 Object：显示从 toString () 返回的字符串	所有的对话框
MessageType (int)	定义对话框的风格，包括布局和使用图标。 对话框风格与界面样式有关。可允许的值是： JOptionPane.ERROR_MESSAGE JOptionPane.INFORMATION_MESSAGE JOptionPane.WARNING_MESSAGE JOptionPane.QUESTION_MESSAGE JOptionPane.PLAIN_MESSAGE	所有的对话框
icon (Icon)	一个装饰图标。如果没有显式地设置，则所使用的图标由消息类型参数来决定	所有对话框
title (String)	对话框的标题	
optionType (int)	定义要在选项窗格中显示的按钮。允许的值是： JOptionPane.DEFAULT_OPTION JOptionPane.YES_NO_OPTION JOptionPane.YES_NO_CANCEL_OPTION JOptionPane.OK_CANCEL_OPTION	确认对话框 选项对话框
selection Values (Object [])	作为输入对话框的组合格或列表中的项使用的一个对象数组。通常，使用最多的是字符串，但是任何对象都可以被作为选择值使用。如果数组中的对象不是字符串，则用从 toString 方法返回的字符串作为项	输入对话框
initial SelectionValue (Object [])	selectionValue 参数中的初始选取对象	输入对话框
options (Object [])	定义在按钮行中显示的对象（重载选项类型）。数据类型决定选项是怎样显示的： Object []：数组中的每个对象都以下面概要的方式进行解释，并按顺序从左到右水平排放在按钮行中。 String []：数组中的每个字符串都以下面概要的方式进行解释，并按顺序从左到右水平排放在按钮行中。 String：作为按钮的标签。 Icon：包裹在一个 JButton 中的图标。 Component：添加到按钮行中的组件	选项对话框
initialValue (Object)	当显示对话框时，按钮行中最初有焦点的组件。	选项对话框

我们知道，在包含选项窗格的对话框中显示的消息可以是任何类型的对象，它们可以是字符串、组件和图标。同样，在按钮行中显示的对象（由 option 属性指定）也可以是任何类型的对象，包括一个对象数组，这些对象按它们在数组中出现的顺序从左到右的水平显示。

可以为由 `static JOptionPane` 方法创建的所有类型的对话框指定父组件、消息、消息类型和对话框标题。

可以为确认对话框和选项对话框指定选项类型参数，这个参数决定在对话框按钮行中显示的按钮集。消息对话框总是只包含一个标签为 OK 的按钮，输入对话框总是包含两个按钮，它们的标签分别是 OK 和 Cancel，因此，不可以为消息对话框或输入对话框指定选项类型参数。

由于输入对话框是唯一支持从一个列表中选择值的对话框类型，所以，选取值和初始选取值参数只能在输入对话框中使用。

因为只有选项对话框才可以定制它们的按钮行，所以，只可以为选项对话框指定选项参数，这个参数用于定制包含在对话框按钮行中的组件。同理，只可以为选项对话框指定初始值参数，这个参数确定按钮行中初始接收焦点的组件。

14.3.3 消息对话框

消息对话框显示一个信息消息，而且总是只有一个标签为 OK 的按钮。消息、标题、图标和对话框的消息类型都是可以设置的。

下面列出了用于创建消息对话框的 `static JOptionPane` 方法。

```
public static void showMessageDialog (Component parentComponent, Object message)
public static void showMessageDialog (Component parentComponent,
                                     Object message, String title, int messageType)
public static void showMessageDialog (Component parentComponent,
                                     Object message, String title, int messageType,
                                     Icon icon)
```

表 14-6 总结了传送给 `JOptionPane showMessageDialog` 方法的这些参数。有关表 14-6 所列的参数的更多信息，请参见表 14-5。

表 14-6 创建消息对话框时所使用的参数

属性	允许值	缺省
<code>parentComponent</code>	任何可见的组件	必须指定
<code>message</code>	一个对象	必须指定
<code>title</code>	一个字符串	"message"
<code>messageType</code>	ERROR_MESSAGE INFORMATION_MESSAGE PLAIN_MESSAGE QUESTION_MESSAGE WARNING_MESSAGE	INFORMATION_MESSAGE
<code>icon</code>	一个图标	根据 <code>messageType</code> ，由界面样式选取

图 14-8 示出的对话框代表五种消息类型：信息、警告、错误、询问和纯文本。这五种消息类型是可以用于 `JOptionPane` 对话框的。图 14-8 示出的对话框是用 Metal 界面样式配置的。

为对话框指定的消息类型决定在对话框中显示的图标（假设没有显式地指定图标）。标准 Swing 界面样式只修改在选项窗格中显示的图标，但其他界面样式还可以修改对话框的布局等其他特性。

图 14-9 所示的小应用程序负责创建和显示图 14-8 示出的对话框。

这个小应用程序包含一个用于选取消息类型的组合框和一个用于创建和显示具有所选消息类型的消息对话框。

这个组合框包含在 `ControlPanel` 的一个实例中，这个实例是 `JPanel` 类的一个扩展。添加到这个组合框中的选项监听器根据在组合框中所选的选项来设置接下来要创建的对话框的消息类型。

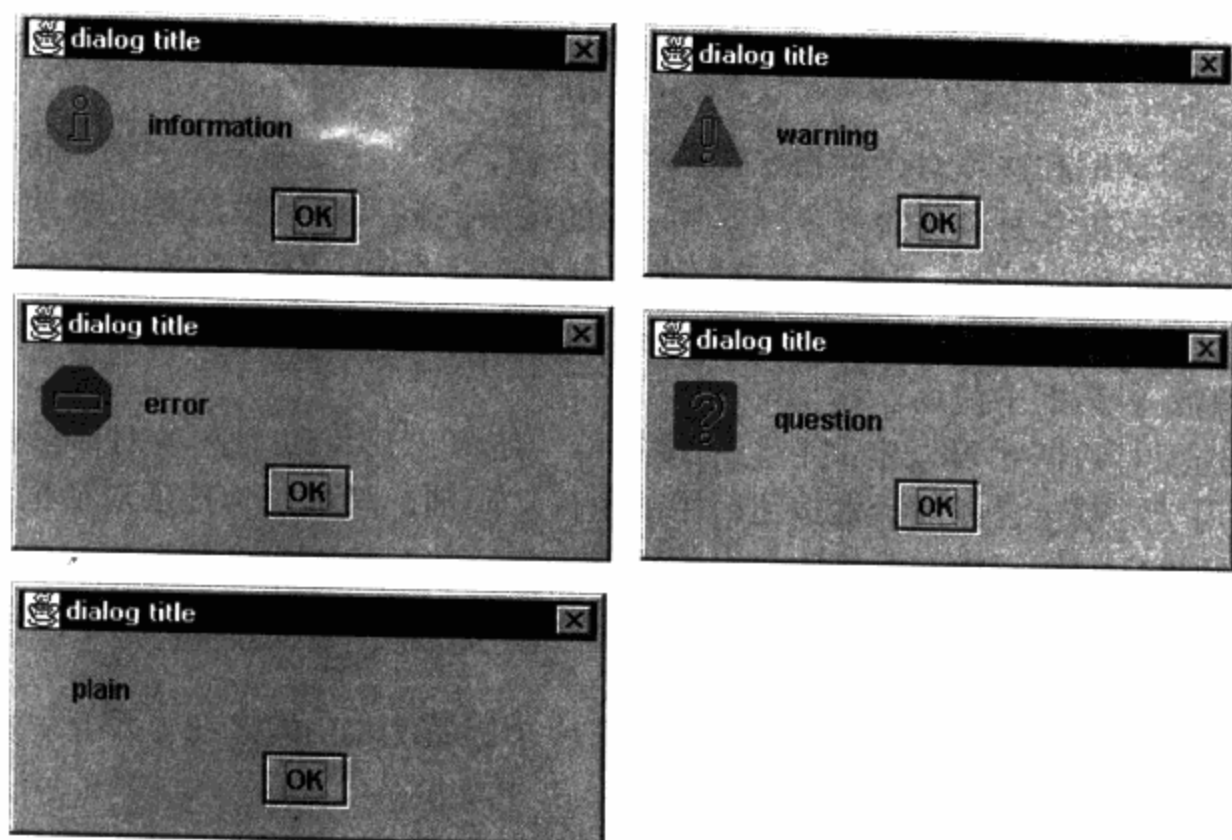


图 14-8 Metal 界面样式的消息类型

```

class ControlPanel extends JPanel {
    private JComboBox messageTypes = new JComboBox ();
    private int [] typeValues = {
        JOptionPane.INFORMATION_MESSAGE,
        JOptionPane.ERROR_MESSAGE,
        JOptionPane.WARNING_MESSAGE,
        JOptionPane.QUESTION_MESSAGE,
        JOptionPane.PLAIN_MESSAGE,
    };
    private String [] typeNames = {
        "JOptionPane.INFORMATION_MESSAGE",
        "JOptionPane.ERROR_MESSAGE",
        "JOptionPane.WARNING_MESSAGE",
        "JOptionPane.QUESTION_MESSAGE",
        "JOptionPane.PLAIN_MESSAGE",
    };
    public ControlPanel (final Test applet) {
        add (messageTypes);
        for (int i=0; i < typeNames.length; ++i) {
            messageTypes.addItem (typeNames [i]);
        }
        messageTypes.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                String s = (String) messageTypes.getSelectedItem ();
                int type;
                for (int i=0; i < typeNames.length; ++i) {
                    if (s.equals (typeNames [i]))
                        applet.setMessageType (typeValues [i]);
                }
            }
        });
    }
}

```

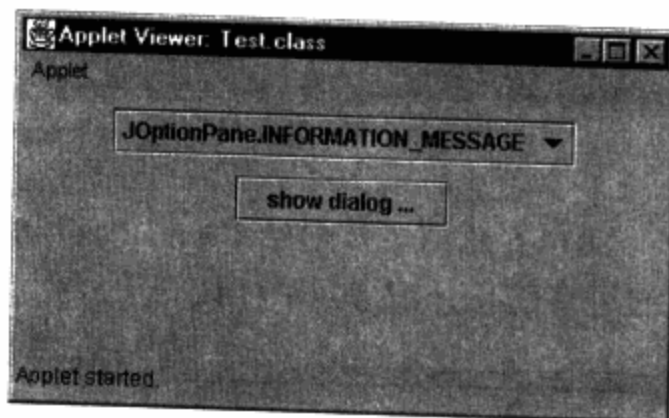


图 14-9 显示不同消息类型的消息对话框

这个小应用程序实现一个 `setMessageType` 方法，与组合框相关联的选项监听器调用这个方法。`setMessageType` 方法以代表消息类型的整数常量为参数，并选取一个要在对话框中显示的相应的信息。

```
...
public void setMessageType (int messageType) {
    this.messageType = messageType;
switch (messageType) {
    case JOptionPane.INFORMATION_MESSAGE:
        message = messages [0];
        break;
    case JOptionPane.ERROR_MESSAGE:
        message = messages [1];
        break;
    case JOptionPane.WARNING_MESSAGE:
        message = messages [2];
        break;
    case JOptionPane.QUESTION_MESSAGE:
        message = messages [3];
        break;
    case JOptionPane.PLAIN_MESSAGE:
        message = messages [4];
        break;
    }
}
```

这个小应用程序把一个动作监听器添加到调用 `JOptionPane.showMessageDialog()` 的按钮中，`JOptionPane.showMessageDialog()` 方法以这个按钮和消息类型为参数。这个按钮用作这个对话框的父组件。

```
public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");
    private String title = "dialog title";
    private String message = "information";
    private int messageType = JOptionPane.INFORMATION_MESSAGE;
    private String messages [] = {
        "information", "error", "warning", "question", "plain"
    };
    public Test () {
        Container contentPane = getContentPane ();
        JPanel controlPanel = new ControlPanel (this);
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (controlPanel);
        contentPane.add (button);
        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JOptionPane.showMessageDialog (
                    button, // parentComponent
                    message, // message
                    title, // title

```

```

        messageType);
    }
    };
}
...

```

例 14-6 列出了图 14-9 所示的小应用程序的代码。

例 14-6 显示具有不同消息类型的消息对话框

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");
    private String title = "dialog title";
    private String message = "information";
    private int messageType = JOptionPane.INFORMATION_MESSAGE;
    private String messages [] = {
        "information", "error", "warning", "question", "plain"
    };

    public Test () {
        Container contentPane = getContentPane ();
        JPanel controlPanel = new ControlPanel (this);
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (controlPanel);
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JOptionPane.showMessageDialog (
                    button, // parentComponent
                    message, // message
                    title, // title
                    messageType);
            }
        });
    }

    public void setMessageType (int messageType) {
        this.messageType = messageType;

        switch (messageType) {
            case JOptionPane.INFORMATION_MESSAGE:
                message = messages [0];
                break;
            case JOptionPane.ERROR_MESSAGE:
                message = messages [1];
                break;
            case JOptionPane.WARNING_MESSAGE:
                message = messages [2];
                break;
            case JOptionPane.QUESTION_MESSAGE:
                message = messages [3];
                break;
            case JOptionPane.PLAIN_MESSAGE:

```

```

        message = messages [4];
        break;
    }
}

class ControlPanel extends JPanel {
    private JComboBox messageTypes = new JComboBox ();
    private int [] typeValues = {
        JOptionPane.INFORMATION_MESSAGE,
        JOptionPane.ERROR_MESSAGE,
        JOptionPane.WARNING_MESSAGE,
        JOptionPane.QUESTION_MESSAGE,
        JOptionPane.PLAIN_MESSAGE,
    };
    private String [] typeNames = {
        "JOptionPane.INFORMATION_MESSAGE",
        "JOptionPane.ERROR_MESSAGE",
        "JOptionPane.WARNING_MESSAGE",
        "JOptionPane.QUESTION_MESSAGE",
        "JOptionPane.PLAIN_MESSAGE",
    };
    public ControlPanel (final Test applet) {
        add (messageTypes);

        for (int i=0; i < typeNames.length; ++i) {
            messageTypes.addItem (typeNames [i]);
        }

        messageTypes.addItemListener (new ItemListener () {
            public void itemStateChanged (ItemEvent e) {
                String s = (String) messageTypes.getSelectedItem ();
                int type;

                for (int i=0; i < typeNames.length; ++i) {
                    if (s.equals (typeNames [i]))
                        applet.setMessageType (typeValues [i]);
                }
            }
        });
    }
}

```

图 14-10 示出的对话框还是一个消息对话框，但它已配备了一个指定的图标。根据与这个选项窗格有关的消息类型，指定的这个图标替换通常由选项窗格的界面样式显示的图标。

例 14-7 列出了创建和显示图 14-10 所示的对话框小应用程序的代码。

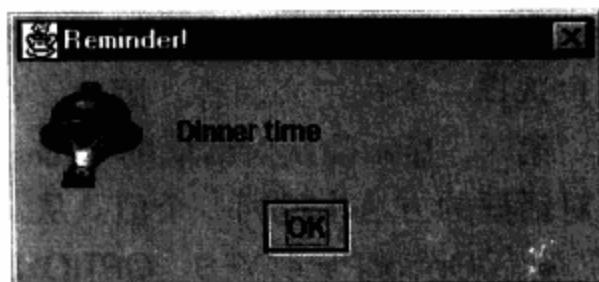


图 14-10 替换消息对话框中的缺省图标

例 14-7 替换消息对话框中的缺省图标

```
import java.awt.*;
```

```
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");

    private String title = "Reminder!";
    private String message = "Dinner time";

    public Test () {
        Container contentPane = getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JOptionPane.showMessageDialog (
                    button, // parentComponent
                    message, // message
                    title, // title
                    JOptionPane.INFORMATION_MESSAGE, // messageType
                    new ImageIcon ("dining.gif")); // icon
            }
        });
    }
}
```

这个小应用程序调用 `JOptionPane.showMessageDialog()` 的版本，这个方法以对一个图标的引用为参数，这个图标在对话框中使用。

Swing 提示

消息类型的解释与界面样式有关

所有的选项窗格都有一个消息类型属性，它为选项窗格指定一个特定的界面样式。标准 Swing 界面样式根据消息类型属性来确定在选项窗格中显示的图标。然而，其他的界面样式还可以根据选项窗格的消息类型属性来修改选项窗格的其他特性。例如，一个定制界面样式可以根据选项窗格的消息类型来修改一个选项窗格的布局。

14.3.4 确认对话框

与消息对话框显示不需要回应的信息消息不同，确认对话框提出一个问题，这个问题需要用户选择一个按钮来回答。因此，在消息对话框和确认对话框之间有两个主要区别。

第一，显示确认对话框的 `static JOptionPane` 方法返回指示激活按钮的 `integer` 值，而显示消息对话框的方法不返回一个值。从 `static JOptionPane` 方法返回的 `integer` 值是下面的常量之一：

- `JOptionPane.YES_OPTION`
- `JOptionPane.NO_OPTION`
- `JOptionPane.CANCEL_OPTION`
- `JOptionPane.OK_OPTION`
- `JOptionPane.CLOSED_OPTION`

除 `JOptionPane.CLOSED_OPTION` 外，上面所列的常量都与激活的按钮相对应。如果关闭这

个对话框，按钮行中的按钮没有一个被激活，例如，用户单击关闭框，则返回 `JOptionPane.CLOSED_OPTION`。

第二，当创建和显示确认对话框时，可以指定 `option type` 属性。这个属性确定要把哪个按钮显示在对话框中，这些按钮在表 14-5 中已介绍过。注意，不能把定制按钮（或其他组件）放在确认对话框的按钮行中。确认对话框只有 Yes 按钮、No 按钮和 Cancel 按钮。

下面列出了创建确认对话框的 `static JOptionPane` 方法。

```
public static int showConfirmDialog (Component parentComponent, Object message)
public static int showConfirmDialog (Component parentComponent, Object message, String title, int optionType)
public static int showConfirmDialog (Component parentComponent, Object message, String title, int optionType, int mes-
sageType)
public static int showConfirmDialog (Component parentComponent, Object message, String title, int optionType,
int messageType, Icon icon)
```

表 14-7 总结了传送给 `JOptionPane showConfirmDialog` 方法的参数。有关表 14-7 所列的参数的详细内容，请参见表 14-5。

表 14-7 创建确认对话框所使用的参数

属性	允许值	缺省
parentComponent	任何可见的组件	必须指定
message	一个对象	必须指定
title	一个字符串	"Select an Option"
messageType	ERROR_MESSAGE INFORMATION_MESSAGE PLAIN_MESSAGE QUESTION_MESSAGE WARNING_MESSAGE	QUESTION_MESSAGE
icon	一个图标	根据 messageType，由界面样式来选择
optionType	YES_NO_OPTION YES_NO_CANCEL_OPTION OK_CANCEL_OPTION	YES_NO_CANCEL_OPTION

图 14-11 所示的小应用程序包含一个按钮，激活这个按钮将创建和显示确认对话框。在激活对话框的一个按钮清除了这个对话框后，被激活的这个按钮的标签显示在这个小应用程序

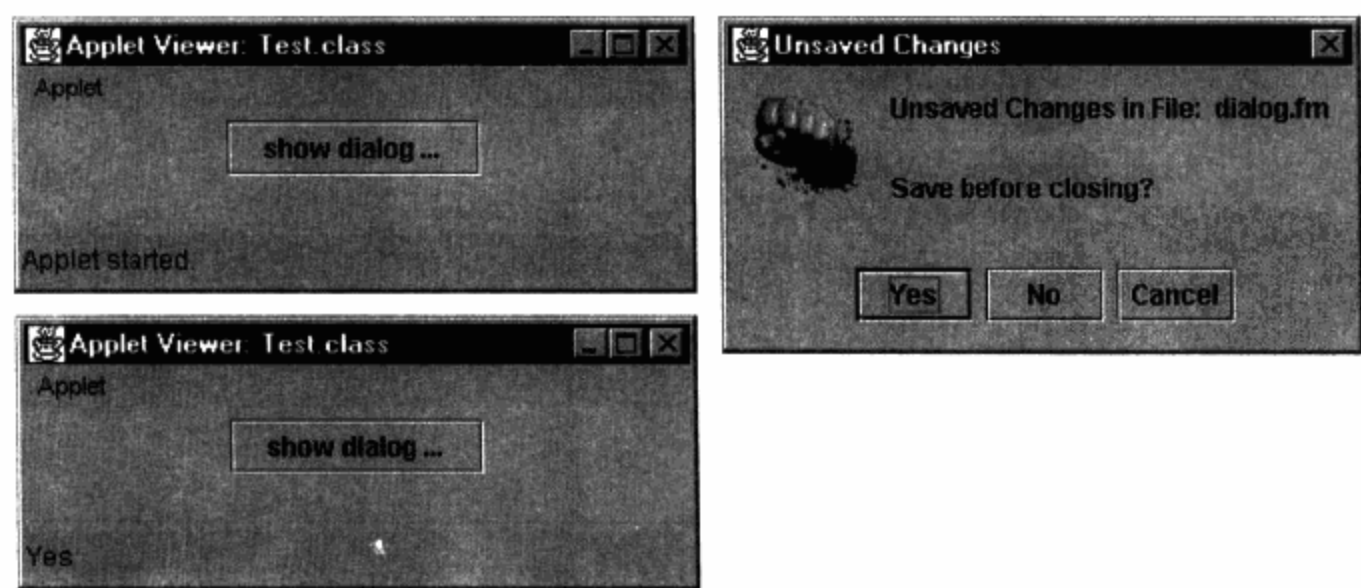


图 14-11 使用确认对话框

的状态区中。

例 14-8 列出了图 14-11 所示的小应用程序的代码。

例 14-8 使用确认对话框

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");

    private String title = "Unsaved Changes";
    private String message [] = {
        "Unsaved Changes in File: dialog.fm",
        "",
        "Save before closing?",
        ""
    };

};

public Test () {
    Container contentPane = getContentPane ();
    contentPane.setLayout (new FlowLayout ());
    contentPane.add (button);

    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            int result = JOptionPane.showConfirmDialog (
                button, // parentComponent
                message, // message
                title, // title
                JOptionPane.YES_NO_CANCEL_OPTION, // optionType
                JOptionPane.WARNING_MESSAGE, // messageType
                new ImageIcon ("punch.gif")); // icon

            switch (result) {
                case JOptionPane.JOptionPaneCLOSED_OPTION:
                    showStatus ("Dialog Closed");
                    break;
                case JOptionPane.YES_OPTION:
                    showStatus ("Yes");
                    break;
                case JOptionPane.NO_OPTION:
                    showStatus ("No");
                    break;
                case JOptionPane.CANCEL_OPTION:
                    showStatus ("Cancel");
                    break;
            }
        }
    });
}
```

指定一个字符串数组来存储在这个对话框中显示的消息。每个字符串按字符串在数组中的出现顺序在这个对话框的消息区中显示。

指定这个对话框的消息类型为 `JOptionPane.WARNING_MESSAGE`，但是消息类型不影响这个对话框的外观，因为显式地指定了一个图标。虽然在这里设置 `WARNING_MESSAGE` 消息类

型对 Metal 界面样式没有什么影响，但是还是要指定这个消息类型，因为可以在一个根据消息类型来修改对话框其他特性的界面样式中使用这个小应用程序。

这个小应用程序根据这个对话框中所激活的按钮来显示相应的字符串。

14.3.5 输入对话框

输入对话框（如它的名字所建议的那样）允许把数据输入到对话框中。指定一个对象数组作为可以选取的值，而且可以指定一个特定的对象作为初始选取值。

通过调用每个对象上的 toString 方法来显示可选取的值。例如，指定作为可选取的值的 JLabel 的实例不能在输入对话框所包含的 combo 框或列表中显示。然而，可以在列表或 combo 框中显示从 JLabel.toString() 返回的字符串。

表 14-8 用于输入对话框的 Swing 组件

组件 ...	当 ... 被用于
JTextField	没有指定可选取的值
JComboBox	指定的可选取的值少于 20 个
JList	指定的可选取的值是 20 或多于 20 个

用来捕获输入的组件与界面样式有关。Swing 标准界面样式提供一个 Swing 组件来显示可选取的值并捕获输入，如表 14-8 所列。

如果没有指定可选取的值，则使用一个文本域。如果指定的可选取的值少于 20 个，则使用 combo 框，如果指定的可选取的值多于 20 个，则使用列表。我们要强调的是：在需要时，可以用定制界面样式来替代其他的组件（或采用完全不同的方法来捕获输入）。

下面列出了用来创建输入对话框的 static JOptionPane 方法。

```
public static String showInputDialog (Object message)
public static String showInputDialog (Component parentComponent, Object message)
public static String showInputDialog (Component parentComponent, Object message, String title, int messageType)
public static objectshowInputDialog (Component parentComponent, Object messageString title, int messageType, Icon icon, Object [] selection Values, Object initialValue)
```

上面所列的前三个 static JOptionPane 方法返回一个代表输入到文本域的文本的字符串。最后一个方法除允许指定一个对象作为初始选取值外，还允许指定一个对象数组作为可选取值，这个方法返回一个代表所选值的 Object 引用。

输入对话框中显示的按钮是不能配置的，输入对话框总是有 OK 按钮和 Cancel 按钮。如果从这些 static showInputDialog 方法中的一个方法返回一个 null 值，则激活了 Cancel 按钮，否则，就是激活了 OK 按钮。

表 14-9 总结了传送给 JOptionPane showInputDialog 方法的参数。有关表 14-9 所列的参数的详细内容，请参见表 14-5。

表 14-9 创建输入对话框要使用的参数

属性	可允许的值	缺省
parentComponent	任何可见的组件	null
message	一个对象	必须指定
title	一个字符串	"Input"
messageType	ERROR_MESSAGE INFORMATION_MESSAGE PLAIN_MESSAGE QUESTION_MESSAGE WARNING_MESSAGE	QUESTION_MESSAGE
icon	一个图标	根据 messageType, 由界面样式选取
selectionValues	一个对象数组	null
initialSelectionValue	一个对象引用	null

注意，当创建输入对话框时，不能指定 `optionType` 属性。即输入对话框的按钮是不可配置的，输入对话框只有一个 OK 按钮和一个 Cancel 按钮。

图 14-12 所示的小应用程序包含一个按钮，激活这个按钮将显示一个输入对话框。没有指定可选取项，因此，用一个文本域来捕获输入。当通过激活 OK 按钮或 Cancel 按钮清除了对话框后，输入文本域的文本将显示在这个小应用程序的状态区中。

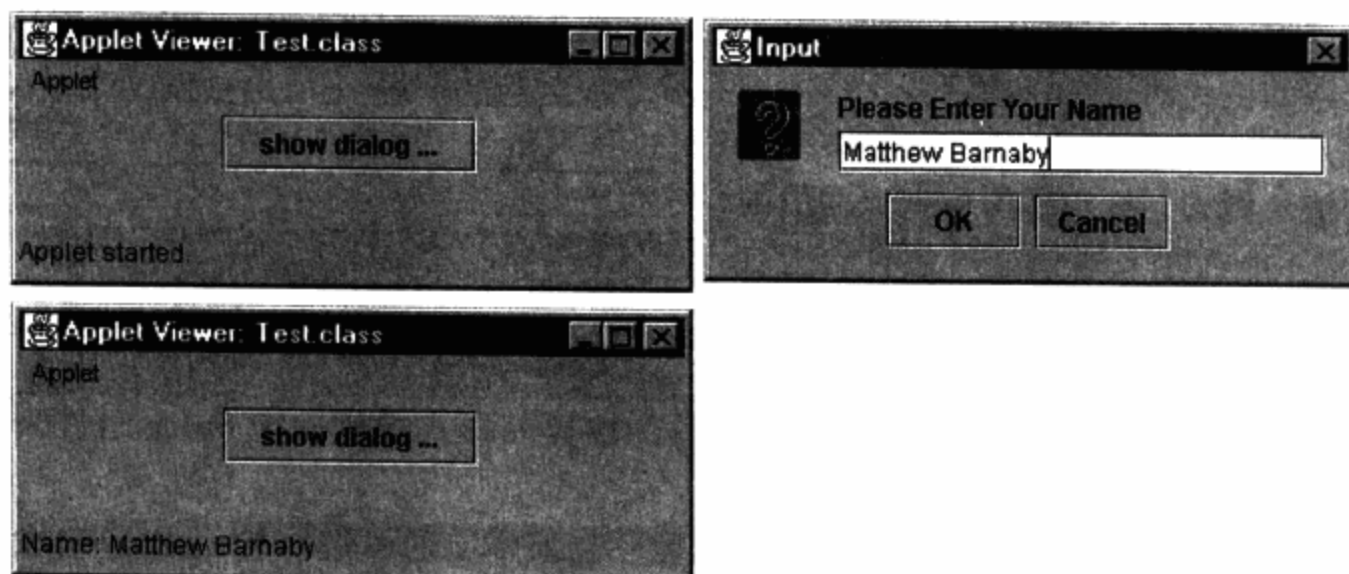


图 14-12 有文本域的输入对话框

例 14-9 列出了图 14-12 所示的小应用程序的代码。

例 14-9 有文本域的输入对话框

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JButton button = new JButton("show dialog ...");
    private String message = "Please Enter Your Name";

    public Test() {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String s = JOptionPane.showInputDialog(message);
                if (s == null)
                    showStatus("cancel button activated");
                else
                    showStatus("Name: " + s);
            }
        });
    }
}
```

这个小应用程序创建一个带简单消息的输入对话框。从 `showInputDialog` 方法中返回一个字符串，并在这个小应用程序的状态区中显示这个字符串。

图 14-13 所示的小应用程序还显示一个输入对话框。因为指定的可选取值小于 20 个，所以这个输入对话框使用一个组合框来显示可选取值。

例 14-10 列出了图 14-13 所示的小应用程序的代码。

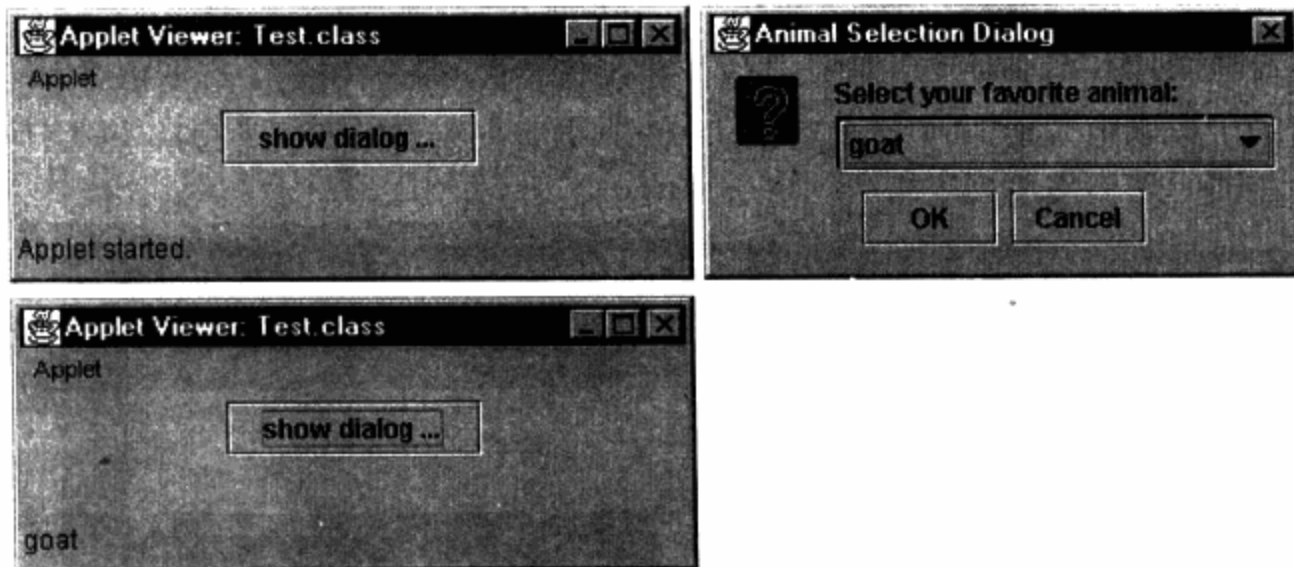


图 14-13 有组合框的输入对话框

例 14-10 有组合框的输入对话框

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");

    private String title = "Animal Selection Dialog";
    private String message = "Select your favorite animal:";
    private String [] selectionValues = {
        "dog", "cat", "mouse", "goat", "koala", "rabbit",
    };

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                String s = (String) JOptionPane.showInputDialog (
                    Test.this, // parentComponent
                    message, // message
                    title, // title
                    JOptionPane.QUESTION_MESSAGE, // messageType
                    null, // icon
                    selectionValues, // selectionValues
                    selectionValues [3]); // initialSelectionValue
                if (s == null)
                    showStatus ("cancel button activated");
                else
                    showStatus (s);
            }
        });
    }
}
```

这个小应用程序除指定可选取值外，还指定 `selectionValues` 数组中的第四项的值作为初始选取值。

这个小应用程序从组合框中选取一个值，并在它的状态区显示这个选取值。

除指定的可选取值大于 20 个以外，图 14-14 所示的小应用程序几乎与图 14-13 所示的小应用程序完全相同。我们使用列表来显示可选取值。

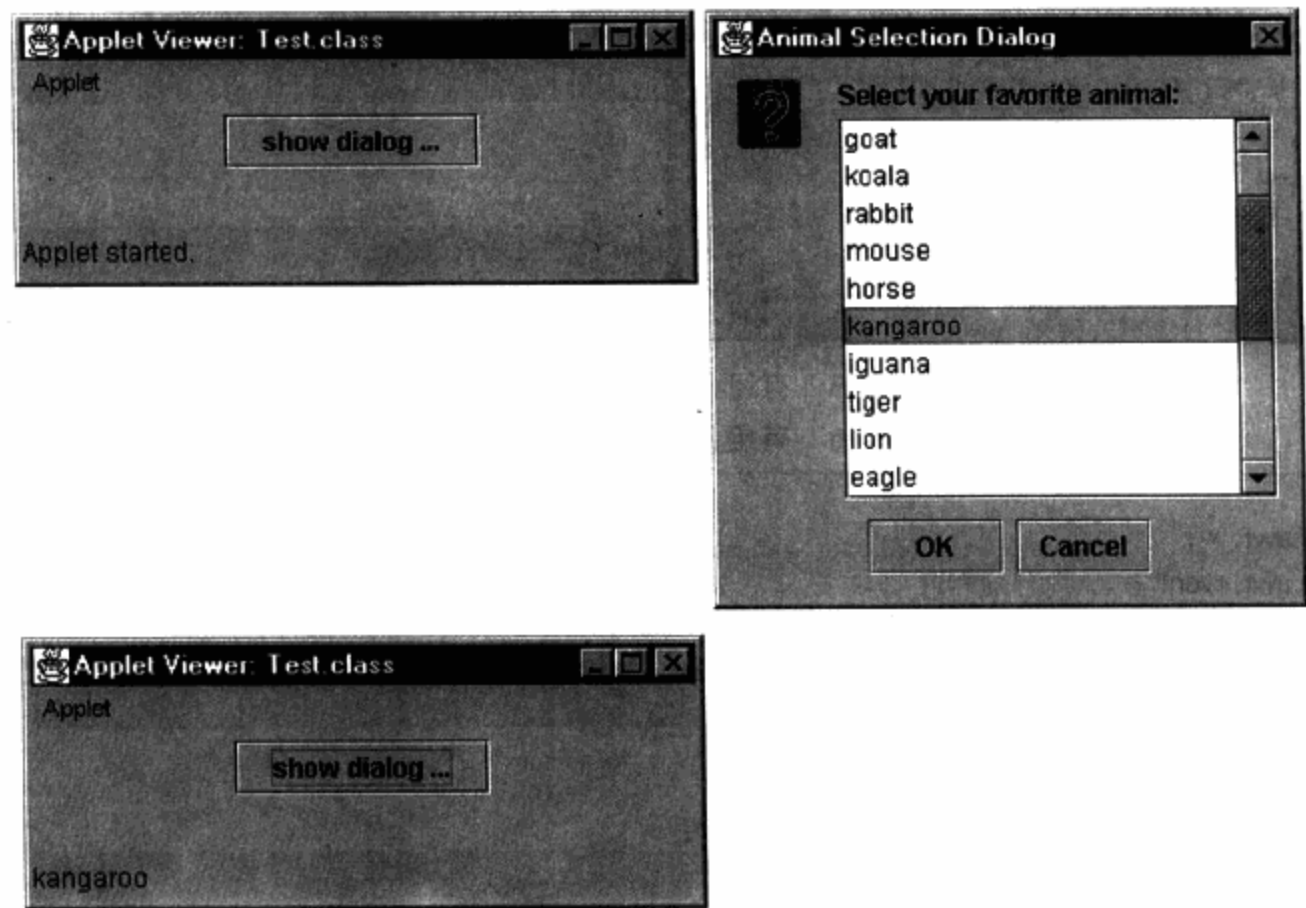


图 14-14 带列表的输入对话框

例 14-11 列出了图 14-14 所示的小应用程序的代码。

例 14-11 带列表的输入对话框

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");

    private String title = "Animal Selection Dialog";
    private String message = "Select your favorite animal:";
    private Object [] selectionValues = {
        "dog", "cat", "mouse", "goat", "koala", "rabbit",
        "mouse", "horse", "kangaroo", "iguana", "tiger", "lion",
        "eagle", "vulture", "wolf", "coyote", "owl", "snake",
        "shrew", "zebra", "wildebeast"
    };

    public Test () {
        Container contentPane = getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
```

```

String s = (String) JOptionPane.showInputDialog (
    Test.this, // parentComponent
    message, // message
    title, // title
    JOptionPane.QUESTION_MESSAGE, // messageType
    null, // icon
    selectionValues, // selectionValues
    selectionValues [3]); // initialSelectionValue

if (s == null)
    showStatus ("cancel button activated");
else
    showStatus (s);
}
});
}

```

Swing 提示

可选取值是作为字符串显示的

输入对话框的可选取值是作为对象来指定的，并且是通过显示由这些对象的 `toString` 方法返回的字符串来显示的。从表 14-5 中可以看到，这与消息类型和选项类型不同，如果它们的实际类型是 `Icon` 或 `JLabel`，则它们作为标签显示；如果消息是一个组件，则按原样显示。消息类型/选项类型的处理方式与可选取值的处理方式不一致的原因是：组合框和列表必须配置定制绘制器以便能显示标签。

如果用字符串以外的对象作为输入对话框的可选取值，则这些对象应该用它们的 `toString` 方法返回的字符串作为可选取值。

14.3.6 选项对话框

选项对话框是唯一一个由 `static JOptionPane` 方法创建的对话框，这个方法提供完全定制对话框按钮行的选项。选项对话框允许在显示对话框时，让按钮行中的组件接收初始焦点。

选项对话框的按钮行用 `options` 参数来定制，接收初始焦点的组件用 `initialValue` 参数来指定。有关 `options` 和 `initialValue` 参数的更多信息，请参见表 14-5。

`JOptionPane` 类提供唯一一个显示选项对话框的方法。

```

public static int showOptionDialog (Component parentComponent,
    Object message,
    String title,
    int optionType,
    int messageType,
    Icon icon,
    Object [] options,
    Object initialValue)

```

表 14-10 总结了传送给 `JOptionPane showInputDialog` 方法的参数。有关表 14-10 所列的参数的更多信息，请参见表 14-5。与 `showConfirmDialog` 方法一样，`showOptionDialog ()` 返回一个指示被激活的按钮的 `integer` 值。

表 14-10 创建选项对话框所使用的参数

属性	允许值	缺省
parentComponent	任何可见的组件	必须指定
message	一个对象	必须指定
title	一个字符串	必须指定
messageType	ERROR_MESSAGE INFORMATION_MESSAGE PLAIN_MESSAGE QUESTION_MESSAGE WARNING_MESSAGE	必须指定
icon	一个图标	必须指定
options	一个对象数组	必须指定
initialValue	一个对象引用	必须指定

图 14-15 所示的小应用程序创建一个选项对话框，它配备了一个带消息的面板和 OK 按钮、Apply 按钮和 Cancel 按钮。在清除这个对话框后，这个小应用程序在它的状态区显示从这个对话框选取的角度。

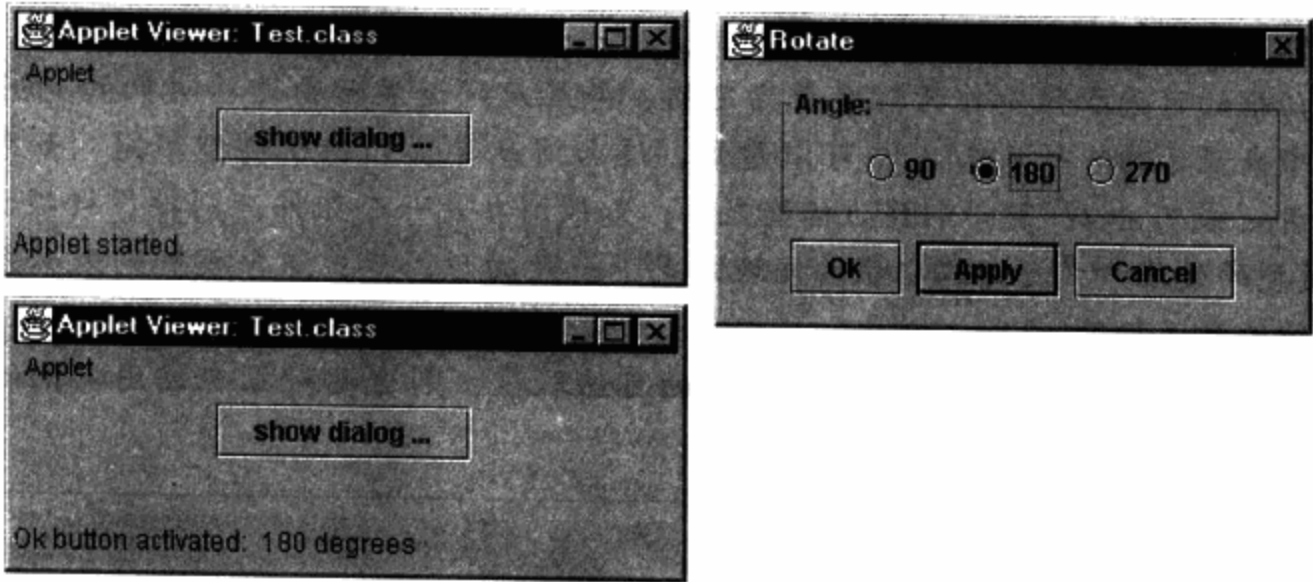


图 14-15 使用选项对话框

这个小应用程序创建 RotatePanel 的一个实例，这个实例是 JPanel 的扩展，它包含一些用于选取角度的单选按钮。旋转面板（RotatePanel）被指定作为这个对话框的消息。因为作为组件指定的消息按原样显示，所以旋转面板显示在这个对话框的消息区中。

这个对话框按钮行中的按钮用一个 Object 数组来指定。这个数组包含两个字符串和一个 JButton 实例，用 showOptionDialog 方法创建的按钮标签对应这两个字符串。只有用 showOptionDialog 方法创建的按钮（在这里是 OK 按钮和 Cancel 按钮）在被激活时才将清除这个对话框。换句话说，对图 14-15 所示的小应用程序，如果激活 OK 按钮和 Cancel 按钮将清除这个对话框，但是激活 Apply 按钮将不能清除这个对话框。

传送给 showOptionDialog 方法的 RotatePanel 和 buttonRowObjects 数组被分别作为这个对话框的消息和选项参数。Apply 按钮还指定作为 initialValue 参数，即当显示这个对话框时，Apply 按钮将接收焦点。

```
public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");
    private JButton applyButton = new JButton ("Apply");
    private RotatePanel rotatePanel = new RotatePanel ();
```



```

private String title = "Rotate";
private Object [] buttonRowObjects = new Object [] {
    "Ok",
    applyButton,
    "Cancel",
};
public Test () {
    ...
    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            int value = JOptionPane.showOptionDialog (
                button, // parentComponent
                rotatePanel, // message
                title, // title
                JOptionPane.DEFAULT_OPTION, //optionType
                JOptionPane.PLAIN_MESSAGE, //messageType, //iconbuttonRowObjects,
                // optionsapplyButton); // initialValue
            ...
        }
    });
}

```

添加到 Apply 按钮中的一个动作监听器将更新这个小应用程序的状态区。JOptionPane.showOptionDialog 方法把两个监听器添加到由这个方法创建的 JButton 的两个实例中；当把这两个按钮激活时，这两个监听器将清除这个对话框。然而，没有把监听器添加到由存在的 JButton 实例指定的按钮中，Apply 按钮就是这种情况。

```

...
applyButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        showStatus (rotatePanel.getSelectedAngle () +
            "degrees");
    }
});
...

```

在清除这个对话框后，根据激活的按钮来更新这个小应用程序的状态区。

```

...
switch (value) {
    case JOptionPane.CLOSED_OPTION:
        showStatus ("Dialog closed with close box");
        break;
    case JOptionPane.OK_OPTION:
        showStatus ("Ok button activated:" +
            rotatePanel.getSelectedAngle () + "degrees");
        break;
    case JOptionPane.CANCEL_OPTION:
        showStatus ("Cancel button activated");
        break;
}
}
}
...

```

例 14-12 列出了图 14-15 所示的小应用程序的代码。

例 14-12 使用一个选项对话框

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;

public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");
    private JButton applyButton = new JButton ("Apply");
    private RotatePanel rotatePanel = new RotatePanel ();
    private String title = "Rotate";
    private Object [] buttonRowObjects = new Object [] {
        "Ok",
        applyButton,
        "Cancel",
    };
}

public Test () {
    Container contentPane = getContentPane ();
    contentPane.setLayout (new FlowLayout ());
    contentPane.add (button);

    applyButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            showStatus (rotatePanel.getSelectedAngle () +
                "degrees");
        }
    });
    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            int value = JOptionPane.showOptionDialog (
                button, // parentComponent
                rotatePanel, // message
                title, // title
                JOptionPane.DEFAULT_OPTION, //optionType
                JOptionPane.PLAIN_MESSAGE, //messageType
                null, //icon
                buttonRowObjects, //options
                applyButton); //initialValue

            switch (value) {
                case JOptionPane.CLOSED_OPTION:
                    showStatus (
                        "Dialog closed with close box"); break;
                case JOptionPane.OK_OPTION:
                    showStatus ("Ok button activated:" +
                        rotatePanel.getSelectedAngle () +
                        "degrees");
                    break;
                case JOptionPane.CANCEL_OPTION:
                    showStatus ("Cancel button activated");
                    break;
            }
        }
    });
}
```

```
class RotatePanel extends JPanel {
    private ButtonGroup group = new ButtonGroup ();

    private JRadioButton [] buttons = {
        new JRadioButton ("0"),
        new JRadioButton ("90"),
        new JRadioButton ("180"),
        new JRadioButton ("270"),
    };

    public RotatePanel () {
        setBorder (BorderFactory.createTitledBorder ("Angle:"));

        for (int i=0; i < buttons.length; ++i) {
            if (i == 0)
                buttons [i].setSelected (true);

            add (buttons [i]);
            group.add (buttons [i]);
        }

        public String getSelectedAngle () {
            String rv = null; // rv = return value

            for (int i=0; i < buttons.length; ++i) {
                if (buttons [i].isSelected ())
                    rv = buttons [i].getText ();
            }

            return rv;
        }
    }
}
```

组件总结 14-3 总结了 JOptionPane 组件。

组件总结 14-3 JOptionPane

模型:	_____
UI 代表:	javax.swing.plaf.BasicOptionPaneUI
绘制器:	_____
编辑器:	_____
激发的事件:	PropertyChangeEvents
替换为:	_____
类图:	见图 14-16

JOptionPane 扩展 JComponent, 而且不实现任何接口。JOptionPane 除定义在属性变化事件被激发时用来访问属性的常量外, 还定义了许多用于设置属性的 public 常量。

JOptionPane 还维护许多 protected 引用, 这些引用跟踪 JOptionPane 的属性 (如消息、初始值、输入值和初始选取值的 Object 引用), 还为选项和选取值属性维护对对象数组的引用。

14.3.7 JOptionPane 属性

表 14-11 列出了由 JOptionPane 类维护的属性。

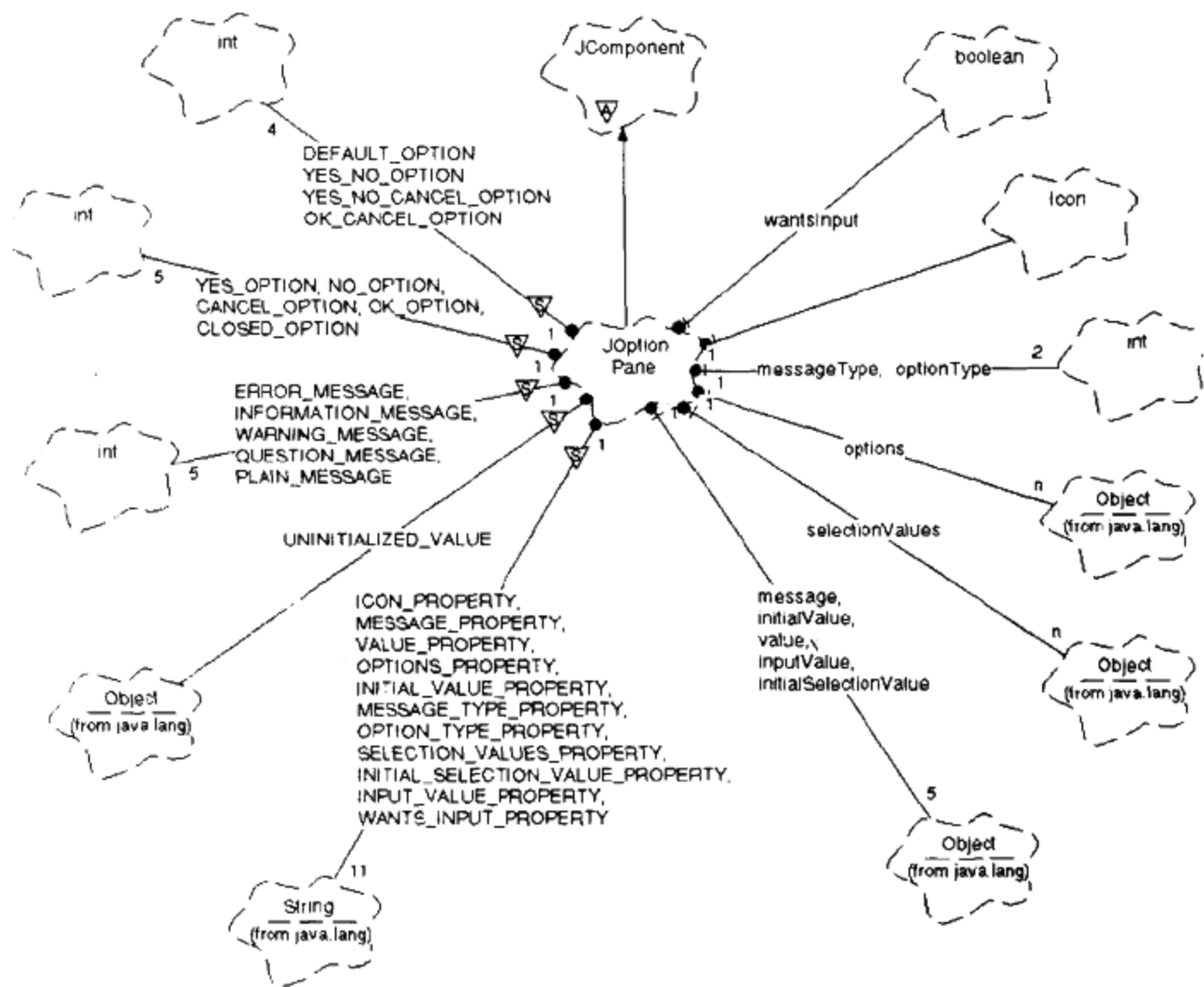


图 14-16 JOptionPane 类图

表 14-11 JOptionPane 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
icon	Icon	B	SG	L&F
initialSelectionValue	Object	B	SG	null
initialValue	Object	B	SG	null
inputValue	Object	B	SG	UNINITIALIZED _ VALUE
maxCharacters	int	S	G	Integer . MAX _ VALUE
PerLineCount				
message	Object	B	SG	"JOptionPane message"
messageType	int	B	SG	PLAIN _ MESSAGE
options	Object []	B	SG	null
optionType	int	B	SG	DEFAULT _ OPTION
selectionValues	Object []	B	SG	null
value	Object	B	SG	UNINITIALIZED _ VALUE
wantsInput	boolean	B	SG	false ^④

① B = 关联的 (激发 `PropertyChangeEvent`) / C = 受约束的 / I = 索引的 / S = 简单的 / Ch = 激发 `ChangeEvent`
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ L&F = 与界面样式有关
④ 当选取值被显式地设置时, 则为 true

icon——在选项窗格中显示的图标。通常，图标在选项窗格的左边显示，但图标的布局最终取决于选项窗格的界面样式。如果没有显式地指定在选项窗格中显示的图标，则界面样式从选项窗格的 `messageType` 属性中确定图标。

initialSelectionValue——用于输入对话框，只是为了选取一个初始值。如果输入对话框使用一个文本域来捕获输入，则当这个对话框显示时，将在文本域中显示这个初始值。如果输入对话框使用一个组合框或一个列表，则这个初始选取值代表组合框或列表中的初始选取值。

initialValue——当选项窗格开始显示时，分配了键盘焦点的一个对象。如果 `initialValue` 对象是 `JButton` 的一个实例，则它是选项窗格的缺省按钮。

inputValue——代表输入对话框中输入值的对象。例如，为一个带文本域的输入对话框设置这个属性将使文本域显示初始的输入值。另外，用 `getInputValue` 方法来获得对对象的一个引用，这个对象是从那些可选取值中选取的。

maxCharactersPerLineCount——一个只读属性，它代表一行消息中可以放置的最大字符数。`JOptionPane` 类从 `getMaxCharactersPerLineCount()` 返回 `Integer.MAX_VALUE`，但是 `JOptionPane` 的扩展可以重载这个方法，如果它们希望限制一行中显示的字符数。

message——`Object` 的一个实例，它相当灵活。要了解不同的数据类型是怎样解释这个属性的，请参见表 14-5。

messageType——由选项窗格的界面样式使用，用来决定在选项窗格中使用的图标和使用的可能的布局。虽然标准界面样式忽略 `messageType` 属性来布局选项窗格，但是定制界面样式可能根据选项窗格的 `messageType` 属性来改变它们的布局。要了解由 `JOptionPane` 类定义的有效 `messageType` 常量，请参见表 14-5。

options——重载 `optionType` 属性并允许把一个定制组件集放在一个选项窗格的按钮行中。这个选项属性的数据类型是一个对象数组。表 14-5 中对这个数组中的每个对象做了解释。

optionType——确定在选项窗格中显示的按钮。这个属性的有效值由 `JOptionPane` 类作为常量来定义，表 14-5 列出了这个属性的有效值。每个值都对应一个标准按钮集，例如，把 `optionType` 属性值设置为 `JOptionPane.YES_NO_CANCEL` 的选项窗格有三个按钮，这三个按钮的标签分别是 Yes、NO 和 Cancel。

为了使用定制按钮，`optionType` 属性可以由选项属性重载，如上面所讨论的那样。

selectionValues——在选项窗格中显示的一个对象数组，选取值是从这个数组中选取的。表 14-5 解释了数组中的每个对象并说明了每个对象的显示方式。

value——代表许多选项中的一个选项，这些选项是对话框按钮行中的那些组件（通常是按钮）。

如果没有选取选项，则 `getValue` 方法返回 `JOptionPane.UNINITIALIZED_VALUE`。

如果用除激活对话框按钮行中的一个按钮以外的其他方式来关闭对话框，则 `getValue()` 返回 `null`。通常这意味着这个对话框是通过用户点击这个对话框的关闭框来关闭的。

wantsInput——由 `JOptionPane` 类使用的一个内部标志，这个属性很少（如果有的话）由开发人员直接操纵。当已经设置了 `selectionValues` 属性时，则这个标志设置为 `true`。

14.3.8 JOptionPane 事件

由 `JOptionPane` 的实例维护的所有属性都是关联属性，即当修改任何属性时，都会激发一个属性改变事件。

图 14-17 所示的小应用程序创建并显示一个包含一个选项窗格的对话框。选项窗格的消息

对象是一个 JPanel，它包含一组复选框。添加到这个选项窗格中的一个属性变化监听器监听 value 属性的变化。当修改 value 属性时（即已经激活了 OK 按钮或 Cancel 按钮），就显示这个值。这个小应用程序用 Windows 界面样式来执行。

这个小应用程序创建 JPanel 的一个实例，这个实例被指定作为这个选项窗格的消息对象。这个选项窗格以一个 QUESTION_MESSAGE 消息类型和一个 OK_CANCEL_OPTION 的选项类型为参数来构造。

把一个动作监听器添加到这个小应用程序的按钮中，这个按钮调用 JOptionPane.createDialog 方法来创建一个对话框。在对话框被显示接着又被清除后，通过调用 JOptionPane.getValue() 来获得这个选项窗格的值。如果从 JOptionPane.getValue() 返回的 Integer 值等于 JOptionPane.OK_OPTION，则调用这个小应用程序的 updateReferences 方法，该方法只在这个小应用程序的状态区显示一个消息。如果这个选项值不等于 JOptionPane.OK_OPTION，则激活 Cancel 按钮并在这个小应用程序的状态区显示一个相应的消息。

```
public class Test extends JApplet {
    ...
    private JPanel messagePanel = new JPanel ();
    ...
    public void init () {
        ...
        final JOptionPane pane = new JOptionPane (
            messagePanel, //message
            JOptionPane.QUESTION_MESSAGE, //messageType
            JOptionPane.OK_CANCEL_OPTION); //optionType
        ...
        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JDialog dialog = pane.createDialog (
                    Test.this, // parentComponent
                    title); // title

                dialog.show (); // blocks

                Integer value = (Integer) pane.getValue ();
                if (value.intValue () == JOptionPane.OK_OPTION)
                    updateReferences ();
                else
                    showStatus ("dialog canceled");
            }
        });
        ...
    }
}
```

把检查属性名是否等于 JOptionPane.VALUE_PROPERTY 的一个属性变化监听器添加到这个选项窗格中。如果这个属性名等于 VALUE_PROPERTY，则显示这个属性的名字和值。

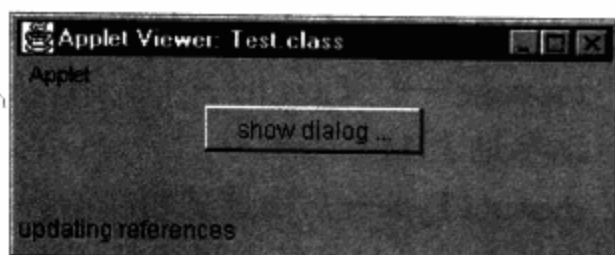
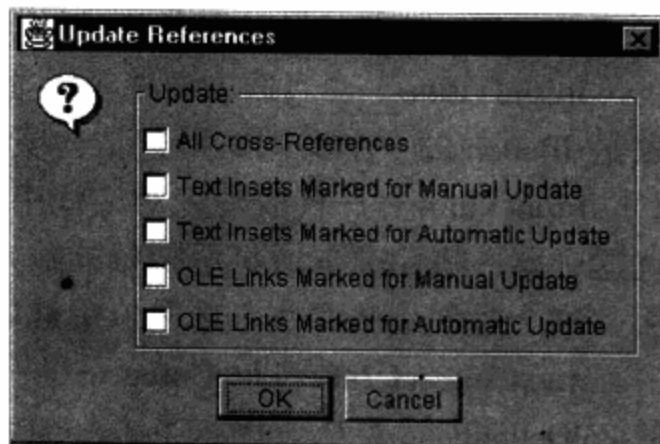
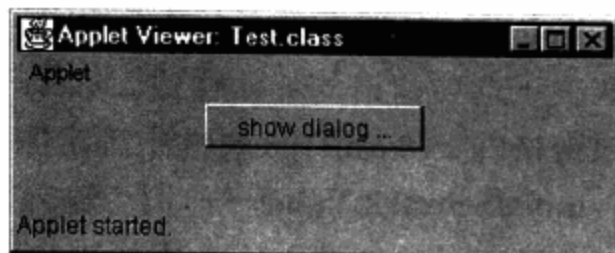


图 14-17 监听 PropertyChangeEvent 事件

```

...
pane.addPropertyChangeListener (
    new PropertyChangeListener () {
        public void propertyChange (PropertyChangeEvent e) {
            String name = e.getPropertyName ();
            if (name.equals (JOptionPane.VALUE _ PROPERTY))
                System.out.println (name + ":" +
                                    e.getNewValue ());
        }
    });
...
}

```

例 14-13 列出了图 14-17 所示的小应用程序的完整代码。

例 14-13 监听从选项窗格激发的 PropertyChangeEvents

```

import java.awt. * ;
import java.awt.event. * ;
import java.beans. * ;
import javax.swing. * ;

public class Test extends JApplet {
    private JButton button = new JButton ("show dialog ...");
    private String title = "Update References";
    private JPanel messagePanel = new JPanel ();
    private JCheckBox [] checkBoxes = {
        new JCheckBox ("All Cross-References"),
        new JCheckBox ("Text Insets Marked for Manual Update"),
        new JCheckBox ("Text Insets Marked for Automatic Update"),
        new JCheckBox ("OLE Links Marked for Manual Update"),
        new JCheckBox ("OLE Links Marked for Automatic Update"),
    };

    public void init () {
        Container contentPane = getContentPane ();
        messagePanel.setBorder (
            BorderFactory.createTitledBorder ("Update:"));
        messagePanel.setLayout (new BoxLayout (messagePanel,
            BoxLayout.Y _ AXIS));

        for (int i=0; i < checkBoxes.length; ++ i)
            messagePanel.add (checkBoxes [i]);

        final JOptionPane pane = new JOptionPane (
            messagePanel, // message
            JOptionPane.QUESTION _ MESSAGE, // messageType
            JOptionPane.OK _ CANCEL _ OPTION); // optionType

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JDialog dialog = pane.createDialog (

```



```

        Test.this, // parentComponent
        title); // title

        dialog.show (); // blocks

        Integer value = (Integer) pane.getValue ();
        if (value.intValue () == JOptionPane.OK_OPTION)
            updateReferences ();
        else
            showStatus ("dialog canceled");
    }
}

pane.addPropertyChangeListener (
    new PropertyChangeListener () {
        public void propertyChange (PropertyChangeEvent e) {
            String name = e.getPropertyName ();
            if (name.equals (JOptionPane.VALUE_PROPERTY))
                System.out.println (name + ":" +
                                    e.getNewValue ());
        }
    });

private void updateReferences () {
    showStatus ("updating references");
}
}

```

14.3.9 JOptionPane 类总结

类总结 14-3 列出了 JOptionPane 的 public 和 protected 变量和方法。

类总结 14-3 JOptionPane

1. 常量

(1) 选项类型

```

public static final int  DEFAULT_OPTION
public static final int  OK_CANCEL_OPTION
public static final int  YES_NO_CANCEL_OPTION
public static final int  YES_NO_OPTION

```

上面所列的这些常量指定 optionType 属性，这个属性确定放置在选项窗格按钮行中的按钮。指定 DEFAULT_OPTION 将导致只有一个 OK 按钮。

(2) 按钮配置选项

```

public static final int  CANCEL_OPTION
public static final int  CLOSED_OPTION
public static final int  NO_OPTION
public static final int  OK_OPTION
public static final int  YES_OPTION

```

上面的这些常量确定选项窗格中的哪个按钮被激活。创建确认对话框的静态 JOptionPane 方法在清除对话框后返回上面所列的常量中的一个常量。可以通过调用 getValue 方法来确定选项窗格中激活的按钮，getValue 方法也返回上面所列的这些常量中的一个常量。

(3) 消息类型

```

public static final int ERROR_MESSAGE
public static final int INFORMATION_MESSAGE
public static final int PLAIN_MESSAGE
public static final int QUESTION_MESSAGE
public static final int WARNING_MESSAGE

```

上面所列的常量指定选项窗格的 messageType 属性，这个属性确定在选项窗格中显示的图标。

(4) 属性

```

public static final String ICON_PROPERTY
public static final String INITIAL_SELECTION_VALUE_PROPERTY
public static final String INITIAL_VALUE_PROPERTY
public static final String INPUT_VALUE_PROPERTY
public static final String MESSAGE_PROPERTY
public static final String MESSAGE_TYPE_PROPERTY
public static final String OPTIONS_PROPERTY
public static final String OPTION_TYPE_PROPERTY
public static final String SELECTION_VALUES_PROPERTY
public static final Object UNINITIALIZED_VALUE
public static final String VALUE_PROPERTY
public static final String WANTS_INPUT_PROPERTY

```

与 JOptionPane 的实例相关联的所有属性都是关联属性 (maxCharactersPerLineCount 除外)，即修改这些属性将激发 PropertyChangeEvent 事件。用上面所列的这些常量来辨别在激发 PropertyChangeEvent 时，由 JOptionPane 的一个实例修改的属性。例如，下面的代码段确定一个 PropertyChangeEvent 是否表示修改了一个选项窗格的消息属性。

```

//code fragment...
anOptionPane.addPropertyChangeListener (
    new PropertyChangeListener () {
        public void propertyChange (PropertyChangeEvent e) {
            String name = e.getPropertyName ();
            if (name.equals (JOptionPane.MESSAGE_PROPERTY)) {
                //property modified is message property
            }
        }
    });

```

2. 构造方法

```

public JOptionPane ()
public JOptionPane (Object message)
public JOptionPane (Object message, int messageType)
public JOptionPane (Object message, int messageType, int optionType)
public JOptionPane (Object message, int messageType, int optionType, Icon icon)
public JOptionPane (Object message, int messageType, int optionType, Icon icon, Object [] options)
public JOptionPane (Object message, int messageType, int optionType, Icon icon, Object [] options,
                    Object initialValue)

```

上面所列的这些构造方法创建各种配置的 JOptionPane 的实例。传送给 JOptionPane 的这些构造方法的参数代表 message、messageType、optionType、icon、options 和 initialValue 属性。要了解在构造 JOptionPane 的实例时可以指定的每个属性的含义，请参见 14.3.7 节“JOptionPane 属性”，要了解缺省值的列表，请参见表 14-11。

图 14-18 示出的小应用程序使用上面所列的大部分构造方法来创建它要显示的选项窗格。这个小应用程序还说明 `JOptionPane` 的实例是组件，这些组件不需要存储在对话框中。

图 14-18 所示的小应用程序中最上面的选项窗格用 `JOptionPane` 无参数的构造方法来构造，这个构造方法创建一个带一个缺省消息和单个 OK 按钮的选项窗格。

```
...
JOptionPane defaultPane = new JOptionPane ();
...
```

下一个选项窗格用以一个对象为参数的构造方法来构造，这个对象代表在选项窗格中显示的消息。此时，这个消息是一个字符串，但是消息可以是任何类型的对象，如表 14-5 所描述的那样。

```
...
JOptionPane messagePane = new JOptionPane
( "Joptionpane (object message)");
...
```

由图 14-18 示出的小应用程序构造的第三个选项窗格指定一个对象数组作为这个选项窗格的消息。这个 `Object` 数组包含三个对象：一个 `JLabel` 实例、一个 `JCheckBox` 实例和一个 `VerboseObject` 实例。`VerboseObject` 类是重载 `toString` 方法的 `java.lang.Object` 的简单扩展。

如何显示对象数组中的每个对象已在表 14-5 中做了介绍。标签和复选框是组件，因此按原样显示。因为 `VerboseObject` 类不是一个组件，所以由它的 `toString` 方法返回的字符串显示在选项窗格的消息区中。

```
...
Object [] objects = new Object [] {
    new JLabel ("JOptionPane (Object message)",
        new ImageIcon ("beach_umbrella.gif"),
        JLabel.LEFT),
    new JCheckBox ("check me out"),
    new VerboseObject (),
};
...
JOptionPane objectPane = new JOptionPane (objects);
...
class VerboseObject extends Object {
    public String toString () {
        return "This is what you'll see in the option pane";
    }
}
```

这个小应用程序构造的第四个选项窗格指定一个消息，这个消息与所使用的 `JOptionPane`

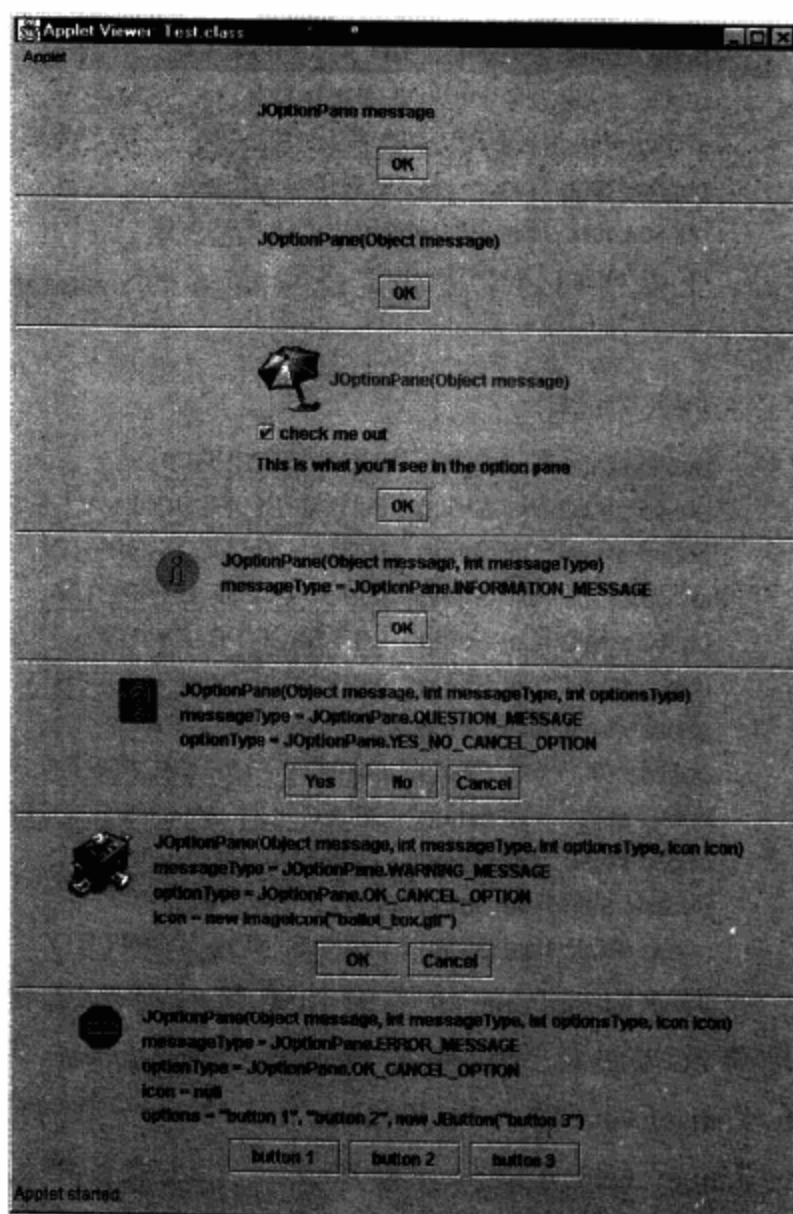


图 14-18 构造各种配置的选项窗格

的构造方法、传送给构造方法的参数和 `JOptionPane.INFORMATION_MESSAGE` 的消息类型有关。注意,由这个小应用程序构造的前三个选项窗格不在选项窗格的图标区中显示一个图标。

如果没有为一个选项窗格显式地指定一个消息类型,则不显示一个图标。这与用 `static JOptionPane` 方法创建的对话框不同,不管是否显式地指定了消息类型,用 `static JOptionPane` 方法创建的对话框都显示一个图标。

```
...
Object [] objects2 = new Object [] {
    "JOptionPane (Object message, int messageType)",
    "messageType = JOptionPane.INFORMATION_MESSAGE",
};
...
JOptionPane messageTypePane = new JOptionPane (
    objects2, JOptionPane.INFORMATION_MESSAGE);
...
```

由这个小应用程序构造的下一个选项窗格指定一个消息、消息类型和选项类型。这个选项类型被指定为 `YES_NO_CANCEL_OPTION`,即按钮行中有 Yes 按钮、No 按钮和 Cancel 按钮。

```
...
Object [] objects3 = new Object [] {
    "JOptionPane (Object message, " +
    "int messageType, int optionsType)",
    "messageType = JOptionPane.QUESTION_MESSAGE",
    "optionsType = JOptionPane.YES_NO_CANCEL_OPTION",
};
...
JOptionPane messageAndOptionsPane = new JOptionPane (
    objects3, JOptionPane.QUESTION_MESSAGE,
    JOptionPane.YES_NO_CANCEL_OPTION);
...
```

由这个小应用程序构造的下一个选项窗格指定一个消息、消息类型为 `WARNING_MESSAGE`、选项类型为 `OK_CANCEL_OPTION` 和一个指定的图标。注意,这个消息类型在选项窗格上是不可见的,因为显式地指定了一个图标。

```
...
Object [] objects4 = new Object [] {
    "JOptionPane (Object message, " +
    "int messageType, int optionsType, Icon icon)",
    "messageType = JOptionPane.WARNING_MESSAGE",
    "optionsType = JOptionPane.OK_CANCEL_OPTION",
    "icon = new ImageIcon ( \"ballot_box.gif \")",
};
...
JOptionPane messageOptionsAndIconPane = new JOptionPane (
    objects4, JOptionPane.WARNING_MESSAGE,
    JOptionPane.OK_CANCEL_OPTION,
    new ImageIcon ("ballot_box.gif"));
...
```

最后一个由这个小应用程序构造的选项窗格指定 `ERROR_MESSAGE` 的消息类型、`OK_CANCEL_OPTION` 的选项类型、一个 `null` 图标和一个 `options` 属性(它是一个包含两个字符串和一个 `JButton` 实例的 `Object` 数组)。

注意,这个选项类型由 `options` 属性重载。虽然这个选项类型指定这个选项窗格应该配备

一个 OK 按钮和一个 Cancel 按钮，但是这个选项类型由 options 属性重载，这个 options 属性显式地指定放置在按钮行中的那些按钮。

```
...
Object [] objects5 = new Object [] {
    JOptionPane (Object message," +
        "int messageType, int optionsType, Icon icon)",
    "messageType = JOptionPane.ERROR_MESSAGE",
    "optionType = JOptionPane.OK_CANCEL_OPTION",
    "icon = null",
    "options = \ "button 1 \ ", \ "button 2 \ ", " +
        "new JButton ( \ "button 3 \ ")");
};
...
Object [] options = {
    "button 1", "button 2", new JButton ("button 3"),
};
JOptionPane messageOptionIconAndOptionsPane =
    new JOptionPane (
        objects5, JOptionPane.ERROR_MESSAGE,
        JOptionPane.OK_CANCEL_OPTION,
        null,
        options,
        options [2]);
...
```

例 14-14 列出了图 14-18 所示的小应用程序的代码。

例 14-14 构造各种配置的选项窗格

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        Object [] objects = new Object [] {
            new JLabel ("JOptionPane (Object message)",
                new ImageIcon ("beach_umbrella.gif"),
                JLabel.LEFT),
            new JCheckBox ("check me out"),
            new VerboseObject (),
        };
        Object [] objects2 = new Object [] {
            "JOptionPane (Object message, int messageType)",
            "messageType = JOptionPane.INFORMATION_MESSAGE",
        };
        Object [] objects3 = new Object [] {
            "JOptionPane (Object message, " +
            "int messageType, int optionsType)",
            "messageType = JOptionPane.QUESTION_MESSAGE",
            "optionType = JOptionPane.YES_NO_CANCEL_OPTION",
        };
        Object [] objects4 = new Object [] {
```

```

        "JOptionPane (Object message, " +
        "int messageType, int optionsType, Icon icon)",
        "messageType = JOptionPane.WARNING_MESSAGE",
        "optionType = JOptionPane.OK_CANCEL_OPTION",
        "icon = new ImageIcon ( \"ballot_box.gif \")",
    };
    Object [] objects5 = new Object [] {
        "JOptionPane (Object message, " +
        "int messageType, int optionsType, Icon icon)",
        "messageType = JOptionPane.ERROR_MESSAGE",
        "optionType = JOptionPane.OK_CANCEL_OPTION",
        "icon = null",
        "options = \"button 1 \", \"button 2 \", " +
        "new JButton ( \"button 3 \")",
    };
    JOptionPane defaultPane = new JOptionPane ();
    JOptionPane messagePane = new JOptionPane (
        "JOptionPane (Object message)");
    JOptionPane objectPane = new JOptionPane (objects);
    JOptionPane messageTypePane = new JOptionPane (
        objects2, JOptionPane.INFORMATION_MESSAGE);
    JOptionPane messageAndOptionTypePane = new JOptionPane (
        objects3, JOptionPane.QUESTION_MESSAGE,
        JOptionPane.YES_NO_CANCEL_OPTION);
    JOptionPane messageOptionAndIconPane = new JOptionPane (
        objects4, JOptionPane.WARNING_MESSAGE,
        JOptionPane.OK_CANCEL_OPTION,
        new ImageIcon ("ballot_box.gif"));
    Object [] options = {
        "button 1", "button 2", new JButton ("button 3"),
    };
    JOptionPane messageOptionIconAndOptionsPane =
        new JOptionPane (
            objects5, JOptionPane.ERROR_MESSAGE,
            JOptionPane.OK_CANCEL_OPTION,
            null,
            options,
            options [2]);
    contentPane.setLayout (new BoxLayout (contentPane,
        BoxLayout.Y_AXIS));
    contentPane.add (defaultPane);
    contentPane.add (new JSeparator ());
    contentPane.add (messagePane);
    contentPane.add (new JSeparator ());
    contentPane.add (objectPane);
    contentPane.add (new JSeparator ());
    contentPane.add (messageTypePane);
    contentPane.add (new JSeparator ());
    contentPane.add (messageAndOptionTypePane);
    contentPane.add (new JSeparator ());
    contentPane.add (messageOptionAndIconPane);

```

```

        contentPane.add (new JSeparator ());
        contentPane.add (messageOptionIconAndOptionsPane);
    }
}

class VerboseObject extends Object {
    public String toString () {
        return "This is what you'll see in the option pane";
    }
}

```

1. 方便的静态方法

```

public static JDesktopPane getDesktopPaneForComponent (Component)
public static Frame getFrameForComponent (Component)
public static Frame getRootFrame ()
public static void setRootFrame (Frame)

```

上面所列的这些方法是 JOptionPane 类使用的一些方便的方法，它们是为在其他环境中使用而提供的。其中的 get... 方法递归地遍历传送给它们的组件的容器结构，直到找到一个合适的容器类型。

2. 确认对话框

```

public static int showConfirmDialog (Component, Object)
public static int showConfirmDialog (Component, Object, String, int)
public static int showConfirmDialog (Component, Object, String, int, int)
public static int showConfirmDialog (Component, Object, String, int, int, Icon)

public static int showInternalConfirmDialog (Component, Object)
public static int showInternalConfirmDialog (Component, Object, String, int)
public static int showInternalConfirmDialog (Component, Object, String, int, int)
public static int showInternalConfirmDialog (Component, Object, String, int, int, Icon)

```

上面所列的这些方法创建确认对话框和内部窗体。为了让人们看清楚，传送给这些方法的参数名没有在这里列出，要了解传送给这些方法的参数名，请参见 14.3.4 节“确认对话框”。

与由静态 JOptionPane 方法创建的所有类型的对话框一样，JOptionPane 类还提供了创建内部窗体的完全类似的方法。

3. 输入对话框

```

public static String showInputDialog (Component, Object)
public static String showInputDialog (Component, Object, String, int)
public static Object showInputDialog (Component, Object, String, int, Icon, Object [], Object)
public static String showInputDialog (Object)

public static String showInternalInputDialog (Component, Object)
public static String showInternalInputDialog (Component, Object, String, int)
public static Object showInternalInputDialog (Component, Object, String, int, Icon, Object [], Object)

```

上面所列的这些方法创建输入对话框和内部窗体。有关创建和使用输入对话框的方式的更多信息，请参见 14.3.5 节“输入对话框”。

4. 消息对话框

```

public static void showMessageDialog (Component, Object)
public static void showMessageDialog (Component, Object, String, int)
public static void showMessageDialog (Component, Object, String, int, Icon)

public static void showInternalMessageDialog (Component, Object)
public static void showInternalMessageDialog (Component, Object, String, int)

```



```
public static void showInternalMessageDialog (Component, Object, String, int, Icon)
```

上面所列的这些方法创建消息对话框和内部窗体。有关创建和使用消息对话框的方式的更多信息，请参见 14.3.3 节“消息对话框”。

5. 选项对话框

```
public static int showOptionDialog (Component, Object, String, int, int, Icon, Object [], Object)
```

```
public static int showInternalOptionDialog (Component, Object, String, int, int, Icon, Object [], Object)
```

上面所列的这些方法创建选项对话框和内部窗体。有关创建和使用选项对话框的方式的更多信息，请参见 14.3.6 节“选项对话框”。

6. 创建对话框和内部窗体/图标

```
public JDialog createDialog (Component, String, title)
```

```
public JInternalFrame createInternalFrame (Component parentComponent, String title)
```

```
public Icon getIcon ()
```

上面所列的这些方法从一个选项窗格实例创建对话框和内部窗体。

`createDialog` 方法创建一个 `JDialog` 实例，它使用窗体作为这个对话框的拥有者，这个窗体与 `parentComponent` 相关联。选项窗格被添加到对话框的内容窗格中作为它的居中组件，设置这个对话框的位置，以便这个对话框在 `parentComponent` 上居中。把一个监听器添加到这个对话框中，以确保被指定作为选项窗格初始值的组件在这个对话框显示时接收键盘焦点。

如果传送给 `createInternalFrame` 的 `parentComponent` 在 `JDesktopPane` 的一个实例中，则创建 `JInternalFrame` 的一个实例，并把选项窗格放在这个内部窗体中，这个内部窗体又被添加到桌面窗格中。如果 `parentComponent` 不在 `JDesktopPane` 的一个实例中，则这个方法会弹出一个运行时刻异常信息。

`getIcon` 方法返回与 `JOptionPane` 的实例相关联的图标。

7. 属性访问方法

```
public Object getInitialSelectionValue ()
```

```
public Object getInitialValue ()
```

```
public Object getInputValue ()
```

```
public int getMaxCharactersPerLineCount ()
```

```
public int getMessage ()
```

```
public int getOptionType ()
```

```
public Object [] getOptions ()
```

```
public Object [] getSelectionValue ()
```

```
public JOptionPaneUI getUI ()
```

```
public String getUIClassID ()
```

```
public Object getValue ()
```

```
public boolean getWantsInput ()
```

```
public void setIcon (Icon)
```

```
public void setInitialSelectionValue (Object)
```

```
public void setInitialValue (Object)
```

```
public void setInputValue (Object)
```

```
public void setMessage (Object)
```

```
public void setMessageType (int)
```

```
public void setOptionType (int)
```

```
public void setOptions (Object [])
```

```
public void setSelectionValue (Object [])
```

```
public void setValue (Object)
```

```
public void setWantsInput (boolean)
```

上面所列的这些方法是 `JOptionPane` 属性的访问方法。有关这些属性和如何使用这些属性

的更多信息，请参见“组件总结 14-3”。

8. 初始值

```
public void selectInitialValue ()
```

`selectInitialValue` 方法选取作为选项窗格初始值的组件。`JOptionPane` 把这个任务交给它的 UI 代表。

9. 更新/设置 UI

```
public void updateUI ()
```

```
public void setUI (OptionPaneUI)
```

上面所列的这两个方法可以在大多数有一个 UI 代表的 `JComponent` 扩展中找到。

14.3.10 AWT 兼容

AWT 不提供与 `JOptionPane` 类似的组件。

14.4 本章回顾

`JWindow` 和 `JDialog` 分别是 AWT 的 `Window` 类和 `Dialog` 类的扩展。因此，Swing 窗口和对话框是包含根窗格的重量容器。

本章介绍的最有趣的组件应当是 `JOptionPane`，它简化了创建对话框和获取用户提供的信息的过程。`JOptionPane` 支持消息对话框、确认对话框、选项对话框和输入对话框，它们可以用 `static JOptionPane` 方法创建和显示。另外，可以创建一个选项窗格，它可以在用 `JOptionPane.createDialog` 创建的对话框中显示。

第 15 章 内部窗体和桌面窗格

Swing 提供实现多文档界面 (MDI) 应用程序的一组组件。MDI 应用程序 (如 Microsoft Word 和 Adobe FrameMaker) 是用一个窗口实现的, 这个窗口是应用程序中创建的文档的桌面。

Swing 提供带桌面的 MDI 功能和内部窗体, 其中桌面由 `JDesktopPane` 类表示, 内部窗体由 `JInternalFrame` 类表示。内部窗体在桌面上, 并且可以在桌面内打开、关闭、最大化和图标化。Swing 还提供一个 `DesktopManager` 类, 用这个类来实现桌面上的内部窗体的特定界面样式行为。

15.1 `JInternalFrame`

由于内部窗体是外部窗体的复制品, 所以内部窗体也是窗体。由于它们包含在另一个 Swing 容器中, 所以它们是内部的, 而这个容器通常是一个桌面窗格。

内部窗体边框中所包含的控件与界面样式有关。标准 Swing 界面样式都提供关闭按钮、最大化按钮和最小化按钮, 这可以从图 15-1 的 Metal 界面样式中看到。另外, Metal 界面样式还提供在标题条中的控制条和图标, 如图 15-1 最下面的图片所示。

单击图 15-1 所示的小应用程序中的按钮将产生内部窗体。图 15-1 顶层的窗体被选取, 此时, 这个窗体的边框是增亮的。

例 15-1 列出了图 15-1 所示的小应用程序的代码。

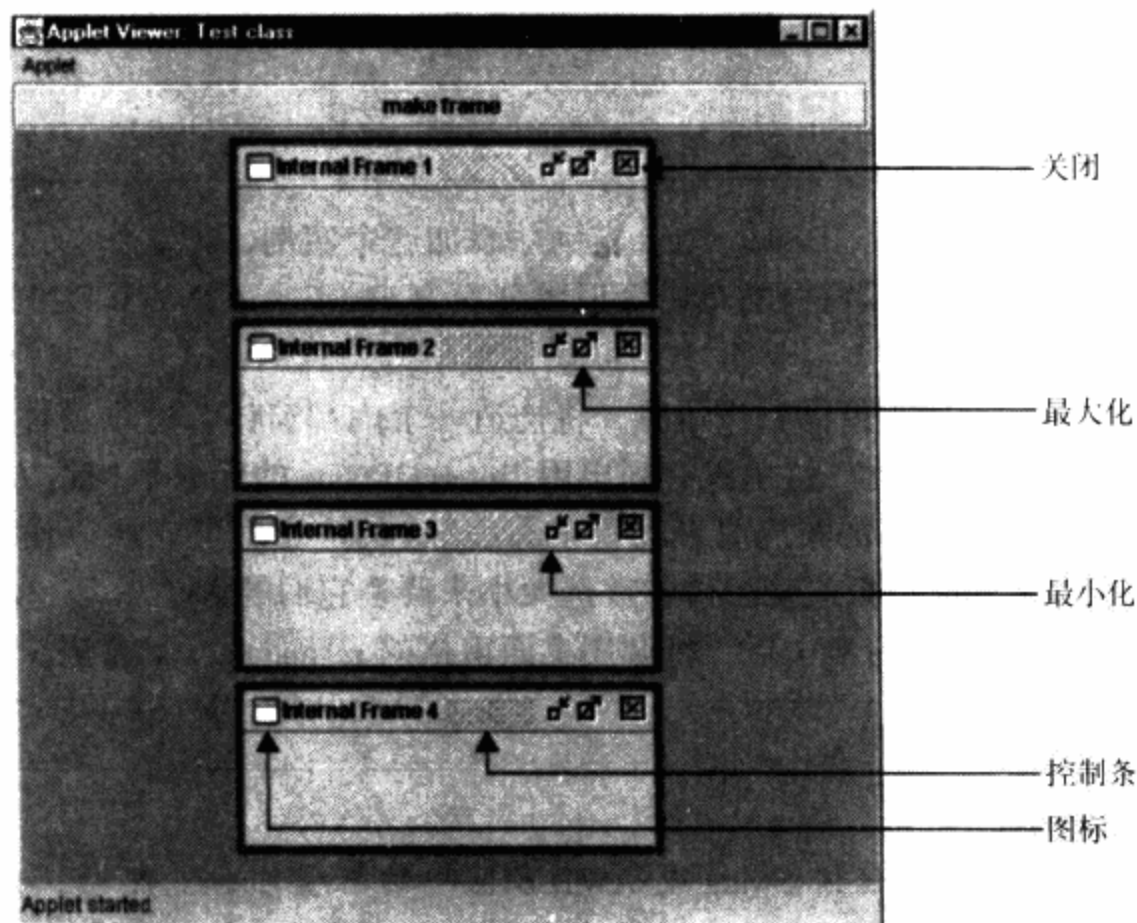


图 15-1 运行中的 `JInternalFrame`

例 15-1 运行中的 JInternalFrame

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class Test extends JApplet {  
    JButton b = new JButton ("make frame");  
    JDesktopPane desktopPane = new JDesktopPane ();  
    int windowCount = 1;  
  
    public void init () {  
        Container contentPane = getContentPane ();  
  
        contentPane.add (b, BorderLayout.NORTH);  
        contentPane.add (desktopPane, BorderLayout.CENTER);  
  
        //JDesktopPane has a null layout manager by default  
        desktopPane.setLayout (new FlowLayout ());  
  
        b.addActionListener (new ActionListener () {  
            public void actionPerformed (ActionEvent event) {  
                JInternalFrame jif = new JInternalFrame (  
                    "Internal Frame " + windowCount + +, // title  
                    true, // resizable  
                    true, // closable  
                    true, // maximizable  
                    true); // iconifiable  
  
                jif.setPreferredSize (new Dimension (250, 100));  
                desktopPane.add (jif);  
                desktopPane.revalidate ();  
            }  
        });  
    }  
}
```

这个小应用程序创建一个 JDesktopPane 实例并添加这个实例，把它作为这个小应用程序内容窗格中的居中组件。把桌面窗格的布局管理器设置为 FlowLayout 的一个实例，因为缺省时桌面窗格的布局管理器是 null。

当激活 make frame (产生窗体) 按钮时，将构造一个内部窗体，这个窗体可调整大小、可关闭、可最大化而且可图标化。这里没有列出用 JInternalFrame 的无参数构造方法构造的 JInternalFrame 实例。因为桌面窗格的布局管理器是 FlowLayout 的一个实例，所以设置了这些内部窗体的首选大小。FlowLayout 根据这些组件的首选大小来调整它们的大小^①。

当添加内部窗体到桌面窗格中后，将调用桌面窗格的 revalidate () 方法。

Swing 提示**桌面窗格中的内部窗体**

内部窗体是可以包含在任何 Swing 或 AWT 容器中的轻量组件。然而，内部窗体希望由 JDesktopPane 的一个实例所包含，例如，如果在内部窗体的容器父组件中没有 JDesktopPane 的一

① 有关布局管理器的更多信息，请参见《Java 2 图形设计，卷 I：AWT》

个实例，则 `JInternalFrame.setMaximum()` 将弹出一个 `NullPointerException` 错误信息。

组件总结 15-1 总结了 `JInternalFrame` 组件。

组件总结 15-1 JInternalFrame

- 模型：——
- UI 代表： `javax.swing.plaf.basic.BasicInternalFrameUI`
- 绘制器：——
- 编辑器：——
- 激发的事件： `PropertyChangeEvents`
- 替换为：——
- 类图：

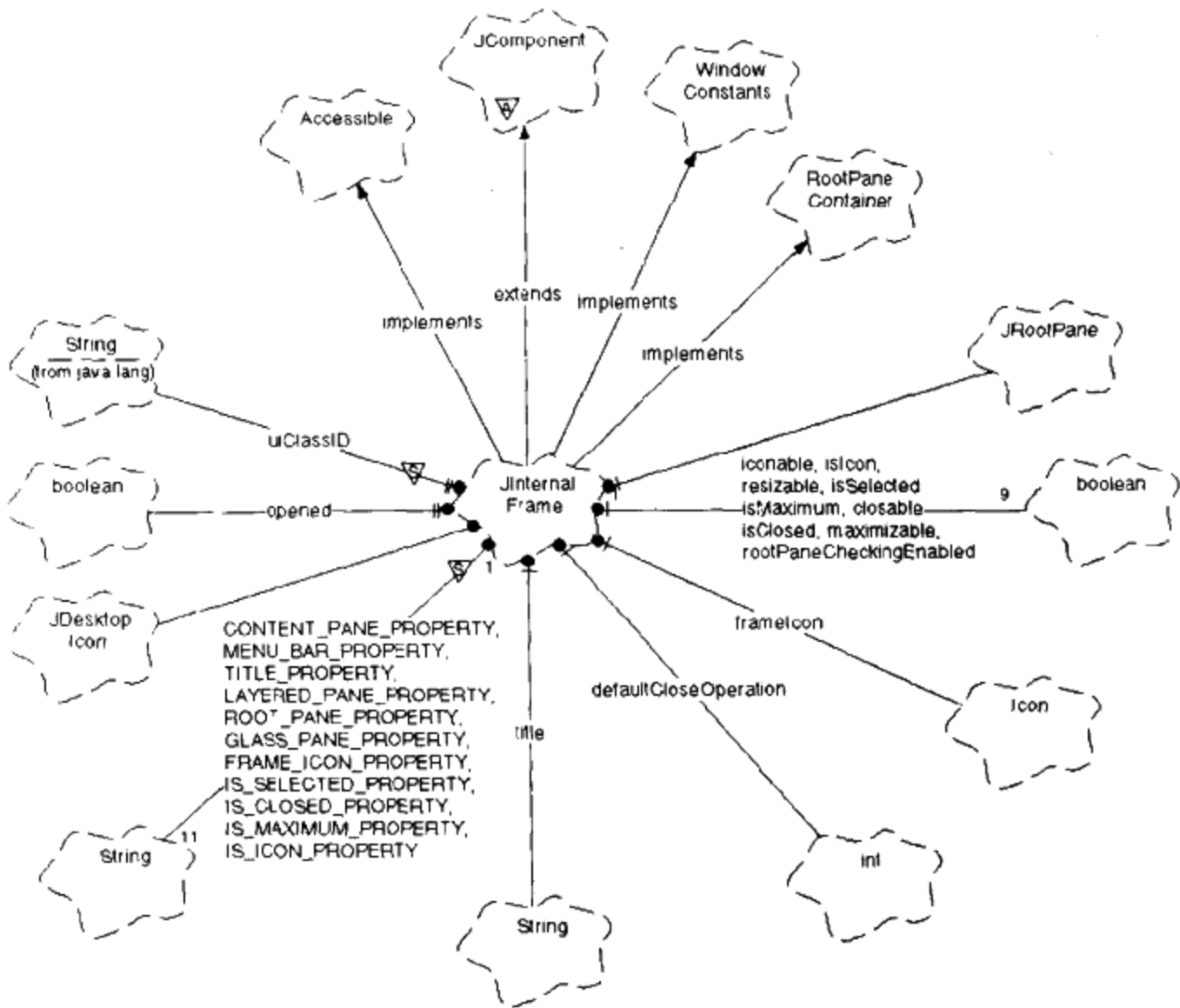


图 15-2 JInternalFrame 类图

`JInternalFrame` 扩展 `JComponent`，并且实现三个接口：`Accessible`、`WindowConstants` 和 `RootPaneContainer`。`JInternalFrame` 维护许多跟踪其状态的 `boolean` 引用，这些状态如窗体是被图标化还是最大化。`JInternalFrame` 还跟踪它的标题、窗体图标和缺省的关闭操作。

15.1.1 JInternalFrame 属性

表 15-1 列出了由 `JInternalFrame` 类维护的属性。

表 15-1 JInternalFrame 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
closable	boolean	S	CSG	false
closed	boolean	CF	SG	false
contentPane	Container	B	SG	JPanel 实例
defaultCloseOperation	int	S	SG	HIDE_ON_CLOSE
desktopIcon	JDesktopIcon	S	SG	L&F
desktopPane	JDesktopPane	——	G	——
frameIcon	Icon	B	SG	L&F
glassPane	Component	B	SG	JPanel 实例
icon	boolean	CBF	SG	false
iconifiable	boolean	S	CSG	false
layer	Integer	S	SG	DEFAULT_LAYER
layeredPane	JLayeredPane	B	SG	JLayeredPane
maximizable	boolean	S	CSG	false
maximum	boolean	CB	SG	false
jMenuBar	JMenuBar	B	SG	null
resizable	boolean	S	CSG	false
rootPane	JRootPane	B	SG	JRootPane
selected	boolean	CBF	SG	false
title	String	B	CSG	null
warningString	String	——	G	null

① B = 关联的 (激发 `PropertyChangeEvent`) / C = 受约束的 / I = 索引的 /
S = 简单的 / Ch = 激发 `ChangeEvent` / F = 激发 `InternalFrameEvent`
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ L&F = 与界面样式有关

closable——指出是否可以通过单击标题条中的关闭按钮等操作来关闭窗口体。closable 属性不影响窗体是否能被程序所关闭。

closed——指出当前是否关闭了窗体。closed 属性是受约束的，即关闭内部窗体和恢复内部窗体可以由一个否决变化监听器否决。

contentPane——与窗体的根窗格相关联的内容窗格。这个内容窗格是容器，它包含窗体中的组件。这些组件应该添加到窗体的内容窗格中，而不是直接添加到窗体中。

defaultCloseOperation——在关闭窗口体时所发生的一个操作。内部窗体的缺省值是 `WindowConstants.HIDE_ON_CLOSE`，与 `JFrame` 和 `JDialog` 的实例一样。有关 `defaultCloseOperation` 属性的详细内容，请参见 14.2.1 节“`JDialog` 属性”。

desktopIcon——当内部窗体被图标化时所显示的一个图标。与内部窗体有关的界面样式负责指定桌面图标。

desktopPane——包含一个或多个内部窗体的一个 `JDesktopPane` 实例。DesktopPane 属性是只读的。

frameIcon——在内部窗体的标题条中显示的图标。如果没有显式地指定这个图标，则由界面样式来指定这个图标。

glassPane——与内部窗体的根窗格相关联的玻璃窗格。有关根窗格和它们的玻璃窗格的详

细内容, 请参见 12.2.2 节“玻璃窗格”。

icon——确定一个内部窗体是否被图标化。调用 `setIcon (true)` 来图标化一个内部窗体, 调用 `setIcon (false)` 将把这个窗体恢复到它原来的位置和大小。`icon` 属性是受约束的, 即可以用一个否决变化监听器来否决图标化和恢复内部窗体。

iconifiable——调用 `setIcon ()` 来确定一个内部窗体是否可以被图标化和恢复。有关图标化和恢复内部窗体的详细内容, 请参见 `icon` 属性。

layer——一个内部窗体所在的层。缺省时, 内部窗体在 `DEFAULT_LAYER` 层上。有关 `JLayeredPane` 类和层的使用方法的详细内容, 请参见 12.3 节“`JLayeredPane`”。

layeredPane——与一个内部窗体的根窗格相关联的层窗格。有关根窗格和它们的层窗格的详细内容, 请参见 12.2 节“`JRootPane`”。

maximizable——指出是否可以通过单击标题条上的图标等操作来最大化一个内部窗体。这个属性不影响一个内部窗体是否可以被程序最大化。

maximum——指出一个内部窗体是否可以被最大化。调用 `setMaximum (true)` 来最大化一个内部窗体, 调用 `setMaximum (false)` 将使这个窗体恢复到它原来的位置和大小。`maximum` 属性是受约束的, 即可以用一个否决变化监听器来否决最大化内部窗体和恢复内部窗体。

jMenuBar——与内部窗体的根窗格相关联的菜单条。有关根窗格和它们的菜单栏的更多信息, 请参见 12.2 节“`JRootPane`”。

resizable——确定是否可以调整一个内部窗体的大小。当内部窗体是最大化的时候, 则不能调整这个内部窗体的大小。

rootPane——与一个内部窗体相关联的根窗格。

selected——选取或取消选取一个内部窗体。通常会绘制被选取的内部窗体, 以便它们以某种形式增亮显示。`Swing` 标准界面样式用与这个窗体的选取状态有关的颜色来绘制所选取的窗体的标题条。`selected` 属性是受约束的, 即可以用一个否决变化监听器来否决选取一个内部窗体和取消选取一个内部窗体。

title——在一个内部窗体的标题条中显示的标题。必须在构造时刻或构造后用 `setTitle (String)` 显式地指定标题, 缺省时, 用一个 `null` 字符串作为内部窗体的标题来构造这个内部窗体。

warningString——用于维护与重量窗体 (`java.awt.Window` 的实例) 的兼容, 当在浏览器中显示重量窗体时, 将在窗口的底部显示一个警告字符串。因为内部窗体总是包含在一个重量窗口中, 所以, 与 `JInternalFrame` 的实例相关联的警告字符串是一个 `null` 字符串, 即内部窗体不需要一个警告字符串。

15.1.2 JInternalFrame 事件

当 `closed`、`icon` 和 `selected` 等属性被修改时, `JInternalFrame` 的实例激发 `InternalFrameEvent` 的实例。另外, `JInternalFrame` 维护下面的受约束属性: `closed`、`icon`、`maximizable` 和 `selected`。

1. 内部窗体事件

无论何时关闭、图标化或选取一个内部窗体, 都把 `InternalFrameEvent` 发送给所有已登记的 `InternalFrameListeners`。接口总结 15-1 总结了 `InternalFrameListeners` 接口。

接口总结 15-1 InternalFrameListeners

```
public abstract void internalFrameActivated (InternalFrameEvent)
public abstract void internalFrameClosed (InternalFrameEvent)
```



```

public abstract void internalFrameClosing (InternalFrameEvent)
public abstract void internalFrameDeactivated (InternalFrameEvent)
public abstract void internalFrameDeiconified (InternalFrameEvent)
public abstract void internalFrameIconified (InternalFrameEvent)
public abstract void internalFrameOpened (InternalFrameEvent)

```

由 InternalFrameListener 接口定义的这些方法在这些方法所代表的事件发生前被调用。例如, 在检测到输入 (如鼠标在窗体中单击) 后, 在实际激活窗体之前立即调用 internalFrameActivated() 方法。作为单个用户行为的结果, 有些由 InternalFrameListener 接口定义的方法被连续调用。例如, 单击一个内部窗体的图标化图标将调用 internalFrameDeactivated(), 再调用 internalFrameIconified()。同样, 恢复一个内部窗体将调用 internalFrameActivated(), 再调用 internalFrameDeiconified()。

类总结 15-1 总结了 InternalFrameEvent 类。

类总结 15-1 InternalFrameEvent

扩展: java.awt.AWTEvent

1. 常量

```
public static final int INTERNAL_FRAME_FIRST
```

```
public static final int INTERNAL_FRAME_LAST
```

上面所列的这些常量代表 InternalFrameEvent 常量中的第一个和最后一个 integer 值。

```
public static final int INTERNAL_FRAME_ACTIVATED
```

```
public static final int INTERNAL_FRAME_CLOSED
```

```
public static final int INTERNAL_FRAME_CLOSING
```

```
public static final int INTERNAL_FRAME_DEACTIVATED
```

```
public static final int INTERNAL_FRAME_DEICONIFIED
```

```
public static final int INTERNAL_FRAME_ICONIFIED
```

```
public static final int INTERNAL_FRAME_OPENED
```

上面所列的这些常量代表 InternalFrameEvent 类所包含的不同类型的事件。这些常量用作事件的 ID, 而且可以用 getID 方法来获得。

2. 构造方法

```
public InternalFrameEvent (JInternalFrame, int id)
```

通常与 Swing (和 AWT) 的事件一样, InternalFrameEvent 类只提供一个构造方法, 这个方法以事件源和一个 integer 值为参数, 这个 integer 值是上面所列的常量之一。

3. 方法

```
public String paramString ()
```

paramString 方法返回一个标识事件的字符串。用 paramString 方法来调试和记录事件。

图 15-3 所示的小应用程序只包含 JInternalFrame 的一个实例。把 InternalFrameListener 的一个实例添加到内部窗体中, 当由 InternalFrameListener 接口定义的方法之一被调用时, 这个窗体就更新这个小应用程序的状态区。左上图显示这个小应用程序开始时的样子。右上图显示在激活内部窗体后, 这个小应用程序的样子。左下图显示在内部窗体被图标化后这个内部窗体的样子。右下图显示在恢复内部窗体后这个内部窗体的样子。

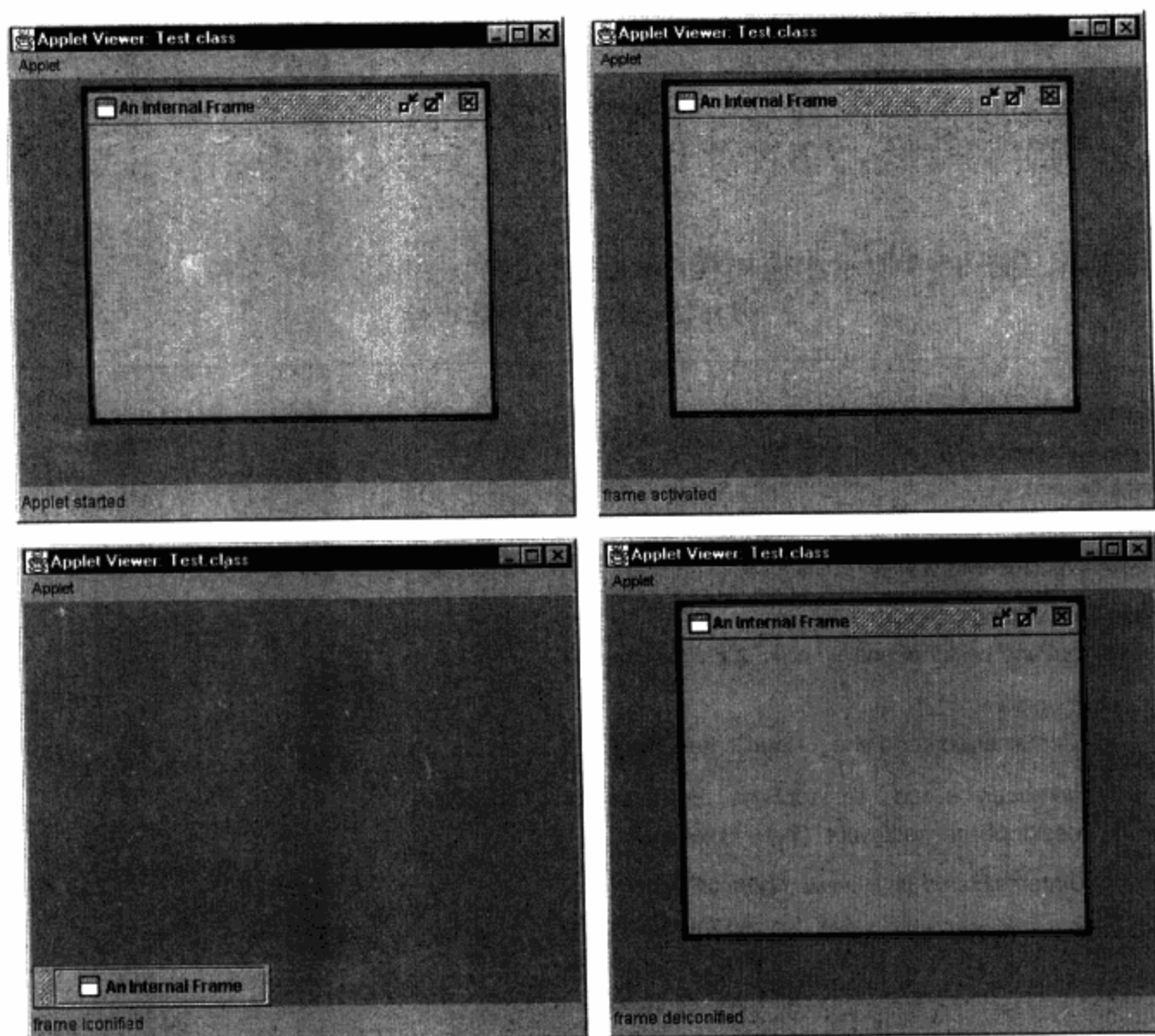


图 15-3 处理 InternalFrameEvent

与这个内部窗体相关联的监听器被列在下面。无论何时调用了这个监听器中的一个方法，这个监听器都只更新这个小应用程序的状态区。

```
class Listener implements InternalFrameListener {
    private JApplet applet;

    public Listener (JApplet applet) {
        this.applet = applet;
    }

    public void internalFrameActivated (InternalFrameEvent e) {
        applet.showStatus ("frame activated");
    }

    public void internalFrameClosed (InternalFrameEvent e) {
        applet.showStatus ("frame closed");
    }

    public void internalFrameClosing (InternalFrameEvent e) {
        applet.showStatus ("frame closing");
    }

    public void internalFrameDeactivated (InternalFrameEvent e) {
        applet.showStatus ("frame deactivated");
    }

    public void internalFrameDeiconified (InternalFrameEvent e) {
        applet.showStatus ("frame deiconified");
    }

    public void internalFrameIconified (InternalFrameEvent e) {
```

```

        applet.showStatus ("frame iconified");
    }
    public void internalFrameOpened (InternalFrameEvent e) {
        applet.showStatus ("frame opened");
    }
}

```

例 15-2 列出了图 15-3 所示小应用程序的完整代码。

例 15-2 处理内部窗体事件

```

import java.awt. * ;
import java.awt.event. * ;
import java.beans. * ;
import javax.swing. * ;
import javax.swing.event. * ;

public class Test extends JApplet {
    JDesktopPane desktopPane = new JDesktopPane ();

    public void init () {
        Container contentPane = getContentPane ();

        contentPane.add (desktopPane, BorderLayout.CENTER);
        desktopPane.setLayout (new FlowLayout ());

        JInternalFrame jif = new JInternalFrame (
            "An Internal Frame", // title
            false, // resizable
            true, // closable
            true, // maximizable
            true); // iconifiable

        jif.setPreferredSize (new Dimension (300, 250));
        jif.addInternalFrameListener (new Listener (this));
        desktopPane.add (jif);
    }
}

class Listener implements InternalFrameListener {
    private JApplet applet;

    public Listener (JApplet applet) {
        this.applet = applet;
    }

    public void internalFrameActivated (InternalFrameEvent e) {
        applet.showStatus ("frame activated");
    }

    public void internalFrameClosed (InternalFrameEvent e) {
        applet.showStatus ("frame closed");
    }

    public void internalFrameClosing (InternalFrameEvent e) {
        applet.showStatus ("frame closing");
    }

    public void internalFrameDeactivated (InternalFrameEvent e) {
        applet.showStatus ("frame deactivated");
    }

    public void internalFrameDeiconified (InternalFrameEvent e) {
        applet.showStatus ("frame deiconified");
    }
}

```

```

|
|
public void internalFrameIconified ( InternalFrameEvent e ) {
    applet.showStatus ( "frame iconified" );
}
|
public void internalFrameOpened ( InternalFrameEvent e ) {
    applet.showStatus ( "frame opened" );
}
|
|

```

2. JInternalFrame 受约束的属性

JInternalFrame 是唯一的、维护受约束属性的 Swing 组件。

与关联属性一样，受约束属性在修改属性时将激发属性变化事件。与关联属性不同的是，可以用任何否决变化监听器来否决受约束属性的变化（所以说是受约束的）。JInternalFrame 把下面这些属性实现为受约束属性：

- closed
- icon
- maximum
- selected

图 15-4 所示的应用程序包含一个可关闭的内部窗体，这个窗体配备了一个 VetoableChangeListener，当关闭这个窗体时，这个 VetoableChangeListener 将进行干预。当这个窗体激发 PropertyChangeEvent 时，监听器调用一个询问变化是否应该被保存的确认对话框。如果激活的是 Cancel 按钮，则监听器将弹出一个 PropertyVetoException 来使变化被否决。

例 15-3 列出了图 15-4 所示的应用程序的代码。

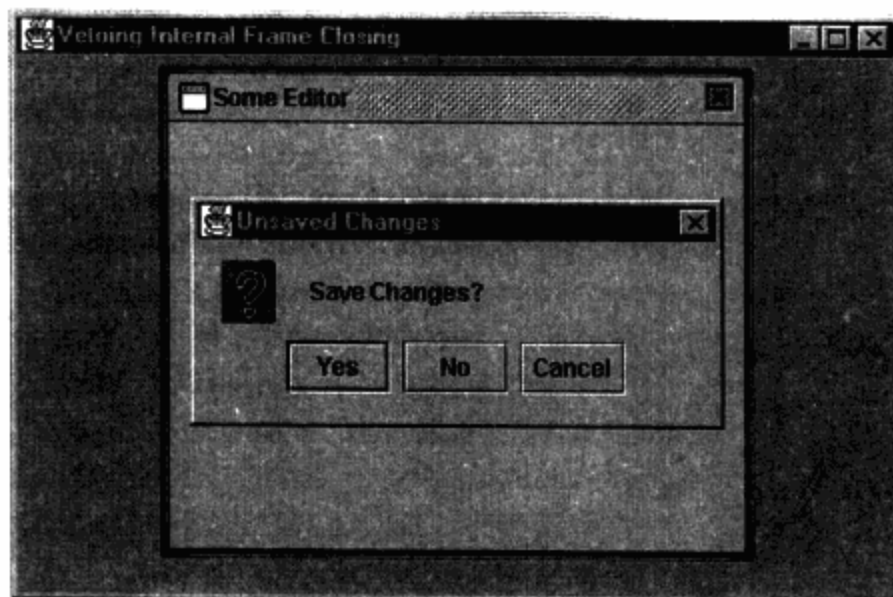


图 15-4 否决关闭内部窗体

例 15-3 否决关闭内部窗体

```

import java.awt. * ;
import java.awt.event. * ;
import java.beans. * ;
import javax.swing. * ;
import java.util. * ;

public class Test extends JFrame {
    JDesktopPane desktopPane = new JDesktopPane ();

    public Test () {
        Container contentPane = getContentPane ();

        contentPane.add ( desktopPane, BorderLayout.CENTER );
        desktopPane.setLayout ( new FlowLayout () );

        JInternalFrame jif = new JInternalFrame (
            "Some Editor", // title
            false, // resizable

```

```

        true); // closable

        jif.setPreferredSize (new Dimension (300, 250));
        jif.addVetoableChangeListener (new CloseListener ());

        desktopPane.add (jif);
    }

    public static void main (String args []) {
        GJApp.launch (new Test (),
            "Vetoing Internal Frame Closing",
            300, 300, 450, 300);
    }
}

class CloseListener implements VetoableChangeListener {
    public void vetoableChange (PropertyChangeEvent e)
        throws PropertyVetoException {
        String name = e.getPropertyName ();

        if (name.equals (JInternalFrame.IS_CLOSED_PROPERTY)) {
            Component internalFrame = (Component) e.getSource ();
            Boolean oldValue = (Boolean) e.getOldValue (),
                newValue = (Boolean) e.getNewValue ();

            if (oldValue == Boolean.FALSE &&
                newValue == Boolean.TRUE) {
                int answer = JOptionPane.showConfirmDialog (
                    internalFrame, // parentComponent
                    "Save Changes?", // message
                    "Unsaved Changes", // title
                    JOptionPane.YES_NO_CANCEL_OPTION);

                if (answer == JOptionPane.CANCEL_OPTION) {
                    throw new PropertyVetoException (
                        "close cancelled", e);
                }
            }
        }
    }
}

```

这个应用程序创建一个 `JDesktopPane` 实例，这个实例被指定为这个应用程序内容窗格的居中组件。然后一个可关闭的 `JInternalFrame` 的实例进行实例化，并把它添加到桌面窗格中。

把一个否决变化监听器（以 `CloseListener` 的一个实例的形式）添加到这个内部窗体中。

`CloseListener` 类实现 `VetoableChangeListener` 并负责显示确认对话框。

`VetoableChangeListener` 接口只定义一个方法——`VetoableChange (PropertyChangeEvent)`，这个方法由 `CloseListener` 类来实现。`VetoableChange` 方法从属性变化事件中获得这个属性名，并检查正在修改的属性是否是 `closed` 属性。如果是，则获得对激发这个事件的内部窗体的一个引用。

从这个属性变化事件中提取这个属性的旧值和新值。只有当旧值是 `false` 而且新值是 `true` 时才显示这个对话框。旧值为 `false` 的含义是这个窗体还没有关闭，新值为 `true` 表示正试图通过把 `closed` 属性值从 `false` 设置为 `true` 来关闭这个窗体。

如果正在修改 `closed` 属性并且这个窗体正在关闭，则调用 `static JOptionPane.showConfirmDialog` 方法来显示确认对话框。如果响应是 `Cancel`，则弹出一个 `PropertyVetoException`。

例 15-3 所列的这个应用程序一个有趣的特点：如果否决内部窗体的关闭，则 `CloseListen-`

er.vetoableChange 方法被调用两次。其原因如下：

可能有其他的监听器监听内部窗体的关闭，而且在 CloseListener 否决这个窗体的关闭前，这些监听器已经收到了变化通知。监听器可能在变化被否决之前对它进行了响应，因此，需要第二个属性变化事件来逆转这个变化。第二个属性变化事件使新值和旧值从原变化逆转过来。

Swing 提示

当否决一个属性时，否决监听器被通知两次

维护受约束属性的类还维护了一个否决监听器列表。当一个调用修改了一个受约束属性的值时，每个否决监听器都通过它的 VetoableChange 方法得到了相应的变化通知，VetoableChange 方法带入一个 PropertyChangeEvent 参数。

如果其中的一个否决监听器通过弹出一个 PropertyVetoException 异常信息来否决这个变化，则用属性的旧值和新值来激发第二个 PropertyChangeEvent 事件，这些旧值和新值与在第一个 PropertyChangeEvent 事件中指定的旧值和新值相反。因为第一个 PropertyChangeEvent 事件可能已经由某些监听器响应了（那些监听器在否决该变化之前认为它是可接受的），所以那些监听器必须有机会取消它们对相应的属性变化所做的任何变化。

类总结 15-2 总结了 JInternalFrame 类。

类总结 15-2 JInternalFrame

1. 构造方法

```
public JInternalFrame ()
public JInternalFrame (String title)
public JInternalFrame (String title, boolean resizable)
public JInternalFrame (String title, boolean resizable, boolean closable)
public JInternalFrame (String title, boolean resizable,
                      boolean closable, boolean maximizable)
public JInternalFrame (String title, boolean resizable, boolean closable,
                      boolean maximizable, boolean iconifiable)
```

检查上面所列的最后一个构造方法的参数列表可以看见在构造时指定的属性列表。

无参数的构造方法构造一个不能调整大小、不能关闭、不能最大化或最小化的内部窗体。

2. 方法

(1) 布局管理器/根窗格/添加组件

```
protected void addImpl (Component, Object, int)
protected JRootPane createRootPane ()
protected boolean isRootPaneCheckingEnabled ()
protected void setRootPaneCheckingEnabled (boolean)
protected void setRootPane (JRootPane)
public void setLayout (LayoutManager)
```

上面所列的前五个方法用于设置窗体的根窗格。与实现 RootPaneContainer 接口的其他 Swing 组件一样，JInternalFrame 不允许把组件直接添加到窗体中，或直接为窗体设置布局管理器。然而，JInternalFrame 本身必须添加根窗格并设置它的布局管理器。因此，setRootPaneCheckingEnabled 和 isRootPaneCheckingEnabled 方法使 JInternalFrame 的实例能添加它们的根窗格并设置它们的布局管理器，同时不允许外部窗体进行这些操作。

createRootPane 方法创建 JRootPane 的一个实例。这个方法是 protected 的，以便 JInternalFrame

的扩展在需要时可以插入一个指定的根窗格中。

setLayout 方法弹出一个异常信息，指出只有这个窗口本身才能设置布局管理器。

(2) 内部窗体事件

```
public synchronized void addInternalFrameListener (InternalFrameListener)
public synchronized void removeInternalFrameListener (InternalFrameListener)
protected void fireInternalFrameEvent ()
```

上面所列的这些方法管理 JInternalFrame 的一个实例的内部窗体监听器。JInternalFrame 的实例使用 protected fireInternalFrameEvent 方法把内部窗体事件发送给内部窗体监听器。

(3) 前景色和背景色

```
public Color getBackground ()
public void setBackground (Color)
public Color getForeground ()
public void setForeground (Color)
```

这些设置和获得 JInternalFrame 实例的背景色的方法可以被重载，以便由内部窗体的内容窗格来设置和获取背景色。

(4) 可否决属性的访问方法

```
public boolean isClosed ()
public boolean isIcon ()
public boolean isMaximum ()
public boolean isSelected ()
public void setClosed (boolean) throws PropertyVetoException
public void setIcon (boolean) throws PropertyVetoException
public void setMaximum (boolean) throws PropertyVetoException
public void setSelected (boolean) throws PropertyVetoException
```

上面所列的这些方法是 JInternalFrame 受约束属性的访问方法。注意，如果我们所谈论的变化由监听器否决，则每个 set... () 方法都可能弹出一个 PropertyVetoException 异常信息。

(5) 属性

```
public Container getContentPane ()
public int getDefaultCloseOperation ()
public JInternalFrame.JDesktopIcon getDesktopIcon ()
public JDesktopPane getDesktopPane ()
public Icon getFrameIcon ()
public Component getGlassPane ()
public int getLayer ()
public JLayeredPane getLayeredPane ()
public JMenuBar getJMenuBar ()
public String getTitle ()
public final String getWarningString ()

public boolean isClosable ()
public boolean isIconifiable ()
public boolean isMaximizable ()
public boolean isResizable ()
public void setClosable (boolean)
public void setContentPane (Container)
public void setDefaultCloseOperation (int)
public void setDesktopIcon (JInternalFrame.JDI)
public void setFrameIcon (Icon)
public void setGlassPane (Component)
public void setIconifiable (boolean)
```



```

public void setLayer (Integer)
public void setLayeredPane (JLayeredPane)
public void setMaximizable (boolean)
public void setJMenuBar (JMenuBar)
public void setResizable (boolean)
public void setTitle (String)
public void setVisible (boolean)

```

上面所列的这些方法是 `JInternalFrame` 类定义的大多数属性的访问方法。在 15.1.1 节“`JInternalFrame` 属性”中介绍了 `JInternalFrame` 属性。

图 15-5 所示的小应用程序显示一个内部窗体，它的图标已经用 `JInternalFrame.setFrameIcon()` 来显式地设置了。

例 15-4 列出了图 15-5 所示的小应用程序的代码。

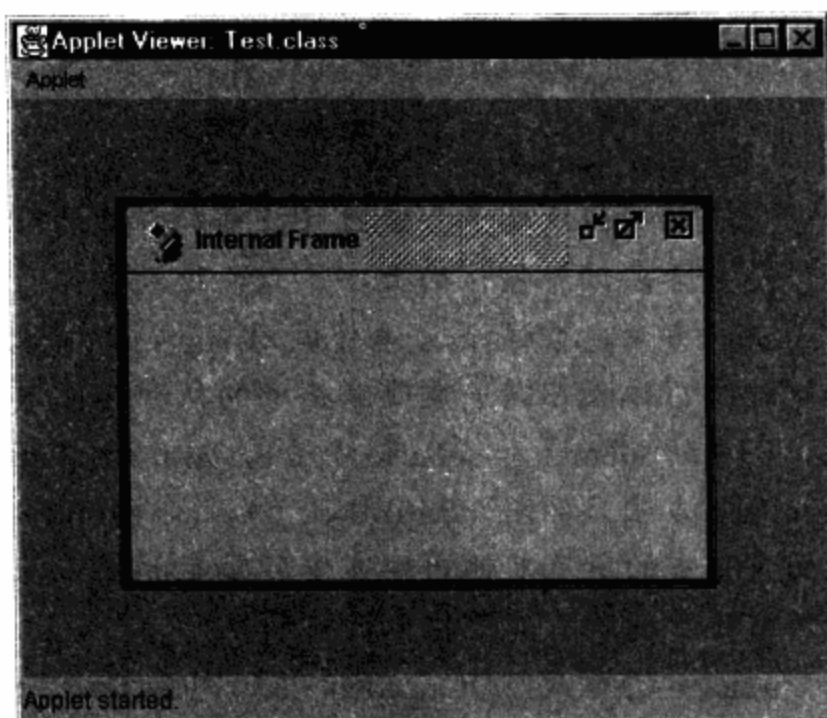


图 15-5 替换一个内部窗体的图标

例 15-4 替换一个内部窗体的图标

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    JDesktopPane desktopPane = new JDesktopPane ();
    JInternalFrame jif = new JInternalFrame (
        "Internal Frame", // title
        true, // resizable
        true, // closable
        true, // maximizable
        true); // iconifiable

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.add (desktopPane, BorderLayout.CENTER);

        jif.setSize (new Dimension (250, 100));
        jif.setFrameIcon (new ImageIcon ("print.gif"));
        desktopPane.add (jif);
    }
}

```

这个小应用程序创建一个桌面窗格和一个内部窗体。这个桌面窗格被添加到这个小应用程序的内容窗格中，而这个内部窗体被添加到桌面窗格中。

在这个窗体的标题条中显示的图标用 `JInternalFrame.setFrameIcon()` 来指定，不要把它与 `JInternalFrame.setIcon()` 混淆起来，`JInternalFrame.setIcon()` 图标化和恢复一个内部窗体。

(6) 显示/前-后台切换/再成形/处理

```

public void show ()
public void toBack ()
public void toFront ()
public void moveToBack ()

```

```
public void moveToFront ()
public void pack ()
public void reshape (int x, int y, int w, int h)
public void dispose ()
```

上面所列的这些方法通过修改内部窗体的可见性、关联性或位置来操纵内部窗体。

`pack` 方法调整窗体的大小，以便这个窗体足够大，能显示它所包含的所有组件。

`show` 方法包装这个窗体并调用 `super.show()`。如果这个窗体已经是可见的，则调用 `show()`把这个窗体放在同一桌面窗格中的所有其他内部窗体的前面。

实现的 `toFront` 和 `toBack` 方法与 `java.awt.Window` 兼容。`toFront` 方法只调用 `moveToFront()`，而 `toBack` 方法只调用 `moveToBack()`。`moveToBack()`改变这个内部窗体的索引，以便它在同一桌面窗格中的所有其他内部窗体后面显示。`moveToFront()` 移动这个内部窗体，以便它在同一桌面窗格中的所有其他窗体的前面显示。

`reshape()`重载了 `JComponent` 类的同名方法，并且在调用 `super.reshape()`后较验和重新绘制这个窗体。

(7) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public InternalFrameUI getUI ()
public String getUIClassID ()
public void setUI (InternalFrameUI)
public void updateUI ()
```

上面列出的方法可以在大多数 `JComponent` 扩展中找到。`Swing` 轻量组件能够返回它们的 `UI` 代表的类名及包含组件的可访问性信息的相关内容。`updateUI` 方法在组件配备了 `UI` 代表时调用。

15.1.3 AWT 兼容

`AWT` 不提供类似内部窗体这样的组件。但是，`JInternalFrame` 类尽量维持一定程度的与 `java.awt.Window` 的兼容性。表 15-2 列出了由 `java.awt.Window` 实现的 `public` 方法和与这些方法相对应的 `JInternalFrame` 方法。

表 15-2 `java.awt.Window` 方法和与 `JInternalFrame` 相对应的方法^①

<code>java.awt.Window</code> 的方法	<code>JInternalFrame</code> 的对应方法
<code>void addWindowListener (WindowListener)</code>	<code>void addInternalFrameListener (WindowListener)</code>
<code>void applyResourceBundle (String)</code>	——
<code>void applyResourceBundle (ResourceBundle)</code>	——
<code>void dispose ()</code>	<code>void dispose ()</code>
<code>Component getFocusOwner ()</code>	——
<code>InputContext getInputContext ()</code>	——
<code>Locale getLocale ()</code>	——
<code>Window [] getOwnedWindows ()</code>	——
<code>Toolkit getToolkit</code>	——
<code>String getWarningString ()</code>	<code>String getWarningString ()</code>
<code>boolean isShowing ()</code>	<code>boolean isVisible ()</code>
<code>void pack ()</code>	<code>void pack ()</code>
<code>void show ()</code>	<code>void show ()</code>
<code>void toBack ()</code>	<code>void toBack ()</code>
<code>void toFront ()</code>	<code>void toFront ()</code>

① 不同的原型用黑体字增亮显示。

除了提供 `java.awt.Window` 与 `JInternalFrame` 之间的兼容等级外，AWT 的 `WindowListener` 的功能与 Swing 的 `InternalFrameListener` 功能相同。表 15-3 列出了由 `java.awt.WindowListener` 接口定义的方法和由 `InternalFrameListener` 定义的对应方法。

表 15-3 java.awt.WindowListener 的方法和 InternalFrameListener 的对应方法

java.awt.WindowListener 方法	JInternalFrame 的对应方法
windowActivated ()	internalFrameActivated ()
windowClosed ()	internalFrameClosed ()
windowClosing ()	internalFrameClosing ()
windowDeactivated ()	internalFrameDeactivated ()
windowDeiconified ()	internalFrameDeiconified ()
windowIconified ()	internalFrameIconified ()

15.2 JDesktopPane

由 `JDesktopPane` 类表示的桌面窗格是包含内部窗体的容器。`JDesktopPane` 是 `JLayeredPane` 类的一个扩展，并且维护与一个桌面管理器的联系。`JDesktopPane` 使用它的层能力来管理它所包含的内部窗体的层序。`JDesktopPane` 的每个实例都有一个负责实现界面样式行为的桌面管理器，在 15.3 节“`DesktopManager`”中将进一步介绍桌面管理器。

`JDesktopPane` 实例一般的使用方式已经在本章介绍的代码样例中说明了。用 `JDesktopPane` 无参数的构造方法来构造 `JDesktopPane` 的实例，这个构造方法是 `JDesktopPane` 提供的唯一的一个构造方法。接着，再把 `JInternalFrame` 的实例添加到这个桌面窗格中。

还可以扩展 `JDesktopPane` 来提供定制的行为。例如，多文档界面（MDI）应用程序的一个普通特性是具有层叠桌面窗格中窗体的能力。图 15-6 所示的小应用程序就具有这个特性，它提供 `JDesktopPane` 的一个扩展，这个扩展可以根据命令来层叠它的窗体。

图 15-6 中的左图显示这个小应用程序开始时的样子。开始时，内部窗体被随机地放置在这个小应用程序的桌面窗格中。提供的 Window 菜单包含三个菜单项：`open all`、`close all` 和 `cascade`。图 15-6 的中间图显示选取 `Cascade` 菜单项后小应用程序的样子，此时，这些在桌面窗格中的内部窗体已经调整大小并重新定位。右图显示在选取 `close all` 菜单项后小应用程序的样子，此时，这些窗体都已经关闭。

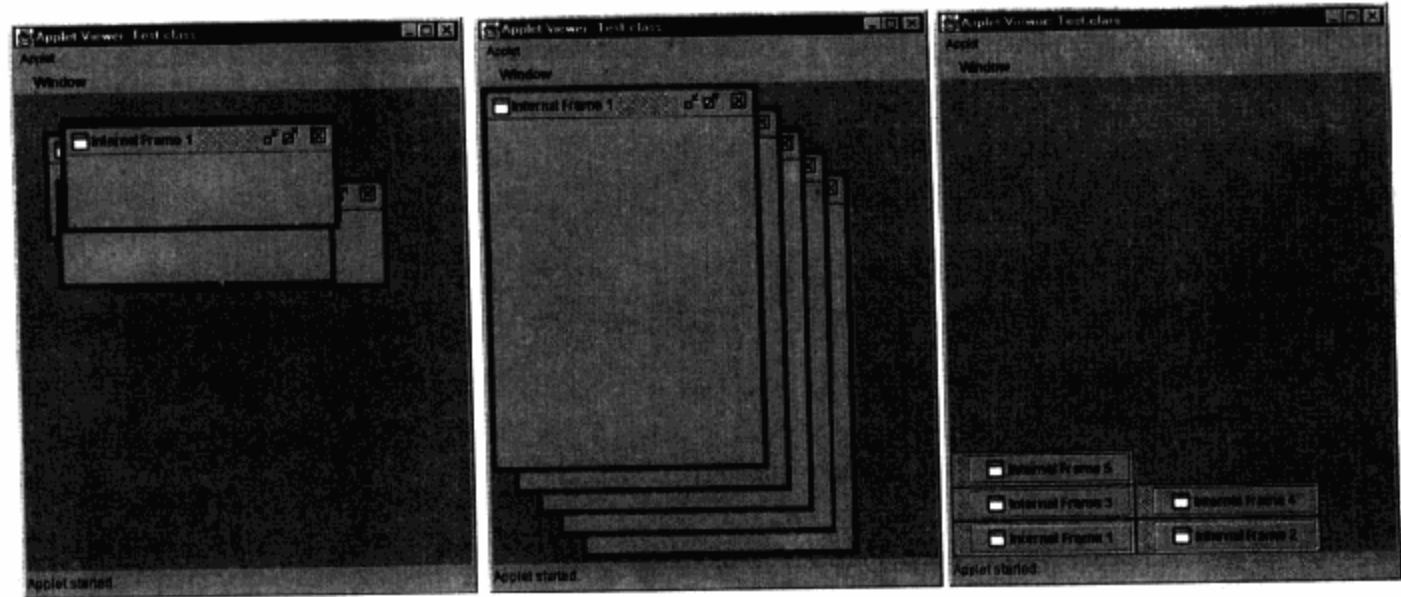


图 15-6 一个定制的桌面窗格

这个小应用程序创建 CustomDesktopPane 的一个实例，并在它上面以随机的位置放置了五个内部窗体。

```
public class Test extends JApplet {
    CustomDesktopPane desktopPane = new CustomDesktopPane ();
    int frameCount = 1, numFrames = 5, x, y;

    public void init () {
        Container contentPane = getContentPane ();

        setJMenuBar (createMenuBar ());
        contentPane.add (desktopPane, BorderLayout.CENTER);

        for (int i=0; i < numFrames; ++i) {
            JInternalFrame jif = new JInternalFrame (
                "Internal Frame" + frameCount + , // title
                true, // resizable
                true, // closable
                true, // maximizable
                true); // iconifiable

            x = (int) (Math.random () * 100);
            y = (int) (Math.random () * 100);

            jif.setBounds (x, y, 250, 100);
            desktopPane.add (jif);
        }
    }
}
```

CustomDesktopPane 类扩展 JDesktopPane，并实现三个方法：closeAll ()、openAll () 和 cascade ()。

```
class CustomDesktopPane extends JDesktopPane {
    private int xoffset = 20, yoffset = 20, w = 250, h = 350;

    public void closeAll () {
        JInternalFrame [] frames = getAllFrames ();

        for (int i=0; i < frames.length; ++i) {
            if (! frames [i] .isIcon ()) {
                try {
                    frames [i] .setIcon (true);
                }
                catch (java.beans.PropertyVetoException ex) {
                    System.out.println ("iconification vetoed!");
                }
            }
        }
    }

    public void openAll () {
        JInternalFrame [] frames = getAllFrames ();

        for (int i=0; i < frames.length; ++i) {
            if (frames [i] .isIcon ()) {
                try {
                    frames [i] .setIcon (false);
                }
                catch (java.beans.PropertyVetoException ex) {
                    System.out.println ("restoration vetoed!");
                }
            }
        }
    }
}
```



```

        desktopPane.add (jif);
    }
}

private JMenuBar createMenuBar () {
    JMenuBar menubar = new JMenuBar ();
    JMenu windowMenu = new JMenu ("Window");

    windowMenu.add (new OpenAllAction ());
    windowMenu.add (new CloseAllAction ());
    windowMenu.add (new CascadeAction ());

    menubar.add (windowMenu);
    return menubar;
}

class OpenAllAction extends AbstractAction {
    public OpenAllAction () {
        super ("open all");
    }

    public void actionPerformed (ActionEvent e) {
        desktopPane.openAll ();
    }
}

class CloseAllAction extends AbstractAction {
    public CloseAllAction () {
        super ("close all");
    }

    public void actionPerformed (ActionEvent e) {
        desktopPane.closeAll ();
    }
}

class CascadeAction extends AbstractAction {
    public CascadeAction () {
        super ("cascade");
    }

    public void actionPerformed (ActionEvent e) {
        desktopPane.cascade ();
    }
}

class CustomDesktopPane extends JDesktopPane {
    private int xoffset = 20, yoffset = 20, w = 250, h = 350;

    public void closeAll () {
        JInternalFrame [] frames = getAllFrames ();

        for (int i=0; i < frames.length; ++i) {
            if (! frames [i] .isIcon ()) {
                try {
                    frames [i] .setIcon (true);
                }
                catch (java.beans.PropertyVetoException ex) {
                    System.out.println ("iconification vetoed!");
                }
            }
        }
    }
}

```

```

public void openAll () {
    JInternalFrame [] frames = getAllFrames ();

    for (int i=0; i < frames.length; ++i) {
        if (frames [i] .isIcon ()) {
            try {
                frames [i] .setIcon (false);
            }
            catch (java.beans.PropertyVetoException ex) {
                System.out.println ("restoration vetoed!");
            }
        }
    }
}

public void cascade () {
    JInternalFrame [] frames = getAllFrames ();
    int x = 0, y = 0;

    for (int i=0; i < frames.length; ++i) {
        if ( ! frames [i] .isIcon ()) {
            frames [i] .setBounds (x, y, w, h);
            x += xoffset;
            y += yoffset;
        }
    }
}

```

Swing 提示

桌面窗格具有 null 布局管理器

缺省情况下, JDesktopPane 的实例有一个 null 布局管理器。因此, 桌面窗格所包含的内部窗体必须显式地指定它们的边界, 否则它们将不能在桌面窗格中显示。

如果为桌面窗格显式地设置了布局管理器, 则这个桌面窗格所包含的内部窗体设置的大小和位置将很可能不起作用, 因为这个布局管理器将重载这些窗体的大小和位置。

例 15-3 所列的小应用程序为包含在这个小应用程序的桌面窗格中的内部窗体设置首选大小, 因为为桌面窗格显式地设置了一个布局管理器, 这个布局管理器根据组件的首选大小来调整组件的大小。相反, 由于桌面窗格有一个 null 布局管理器, 所以图 15-5 所列的小应用程序为包含在这个小应用程序的桌面窗格中的内部窗体设置实际的大小。

组件总结 15-2 总结了 JDesktopPane 组件。

组件总结 15-2 JDesktopPane

模型:	_____
UI 代表:	javax.swing.plaf.basic.BasicDesktopPaneUI
绘制器:	_____
编辑器:	_____
激发的事件:	_____
替换为:	_____

类图： 见图 15-7

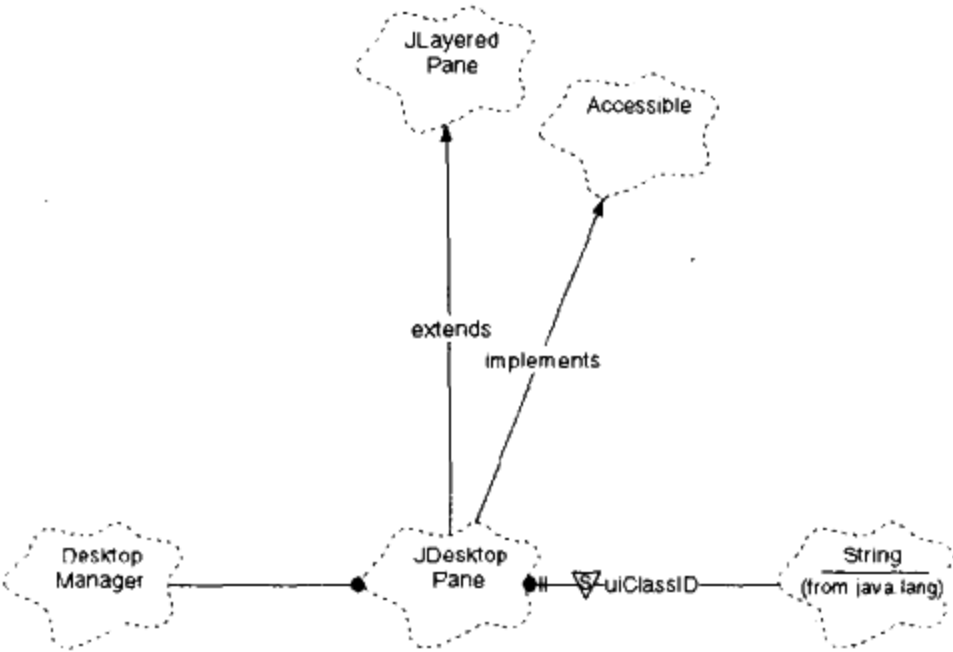


图 15-7 JDesktopPane 类图

JDesktopPane 是 JLayeredPane 的一个简单扩展，它实现 Accessible 接口。每个 JDesktopPane 实例都有它自己的桌面管理器，这个管理器实现桌面窗格所包含的内部窗体的指定界面样式。

15.2.1 JDesktopPane 属性

表 15-4 列出了由 JDesktopPane 类维护的属性。

表 15-4 JDesktopPane 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
desktopManager	DesktopManager	S	SG	DesktopManager
allFrames	JInternalFrame []	—	G	—
allFramesInLayer	JInternalFrame []	I	G	—

- ① B = 关联的 (激发 PropertyChangeEvent) /C = 受约束的 / I = 索引的 /
S = 简单的 /Ch = 激发 ChangeEvent
- ② C = 可在创建时设置 /G = 获取方法 /S = 设置方法
- ③ 1&F = '与界面样式有关'
- desktopManager**——实现与之相关联的桌面窗格的指定界面样式。桌面管理器在 15.3 节 “DesktopManager” 中介绍。
- allFrames**——包含在 JDesktopPane 的一个实例中的所有内部窗体。
- allFramesInLayer**——包含在 JDesktopPane 的一个实例中的所有内部窗体，这些内部窗体在一个特定的层上。

15.2.2 JDesktopPane 事件

JDesktopPane 不激发它自己的任何事件。有关由 JLayeredPane（它是 JDesktopPane 的超类）激发的事件的介绍，请参见 12.3 节 “JLayeredPane”。

15.2.3 JDesktopPane 类总结

类总结 15-3 列出了 JDesktopPane 的 public 和 protected 变量和方法。

类总结 15-3 JDesktopPane

扩展: JLayeredPane

实现: javax.accessibility.Accessible

1. 构造方法

public JDesktopPane ()

JDesktopPane 只提供了一个无参数的构造方法。这个方法构造的桌面窗格带一个桌面管理器, 这个桌面管理器是适合于与这个桌面窗格相关联的界面样式的。

2. 方法

(1) 属性访问方法

public JInternalFrame [] getAllFrames ()

public JInternalFrame [] getAllFramesInLayer (int)

public DesktopManager getDesktopManager ()

public boolean isOpaque ()

public void setDesktopManager (DesktopManager)

上面所列的这些方法是 JDesktopPane 属性的访问方法。通过调用 getAllFrames 或 getAllFramesInLayer 方法可以获得 JInternalFrame 实例的数组。通过分别调用 setDesktopManager 和 getDesktopManager 方法可以设置桌面管理器和获得对当前桌面管理器的一个引用。

JDesktopPane 从 JComponent 类中重载 isOpaque 方法来返回 true, 即所有的桌面窗格都绘制它们所包含的每个像素, 因此, 这些桌面窗格没有透明的背景。

(2) 可访问性/插入式界面样式

public AccessibleContext getAccessibleContext ()

public DesktopPaneUI getUI ()

public String getUIClassID ()

public void setUI (DesktopPaneUI)

public void updateUI ()

上面的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

15.2.4 AWT 兼容

AWT 不提供与 JDesktopPane 类似的组件。

15.3 DesktopManager

桌面管理器实现桌面窗格的特定界面样式行为。例如, 在桌面窗格中拖动内部窗体和调整内部窗体大小的方式由这个桌面窗格的桌面管理器来负责。当一个内部窗体被操纵时, (例如, 通过调用 JInternalFrame.setClosed() 来关闭这个窗体), 这个内部窗体就把操纵工作交给它的 UI 代表, 而 UI 代表又把这个工作交给一个桌面管理器。

桌面管理器负责实现下面的行为:

- 激活窗体/使窗体不激活。
- 拖动窗体/调整窗体的大小。
- 打开窗体/关闭窗体。
- 图标化窗体/恢复窗体。
- 窗体最小化/窗体最大化。

图 15-8 所示的方案图说明了当调用 `JInternalFrame.setClosed()` 时所发生的事件序列。

调用没有被否决的 `JInternalFrame.setClosed()` 将激发一个属性变化事件。这个属性变化事件被传递给内部窗体的 UI 代表^①，这个 UI 代表又把这个事件交给它的桌面管理器。缺省桌面管理器通过获得对桌面窗格（与这个内部窗体相关联）的一个引用，并且调用 `JDesktopPane.remove()` 方法（以要被删除的内部窗体的一个引用为参数）来关闭一个内部窗体。然后，缺省桌面管理器重新绘制受影响的桌面窗格区域。

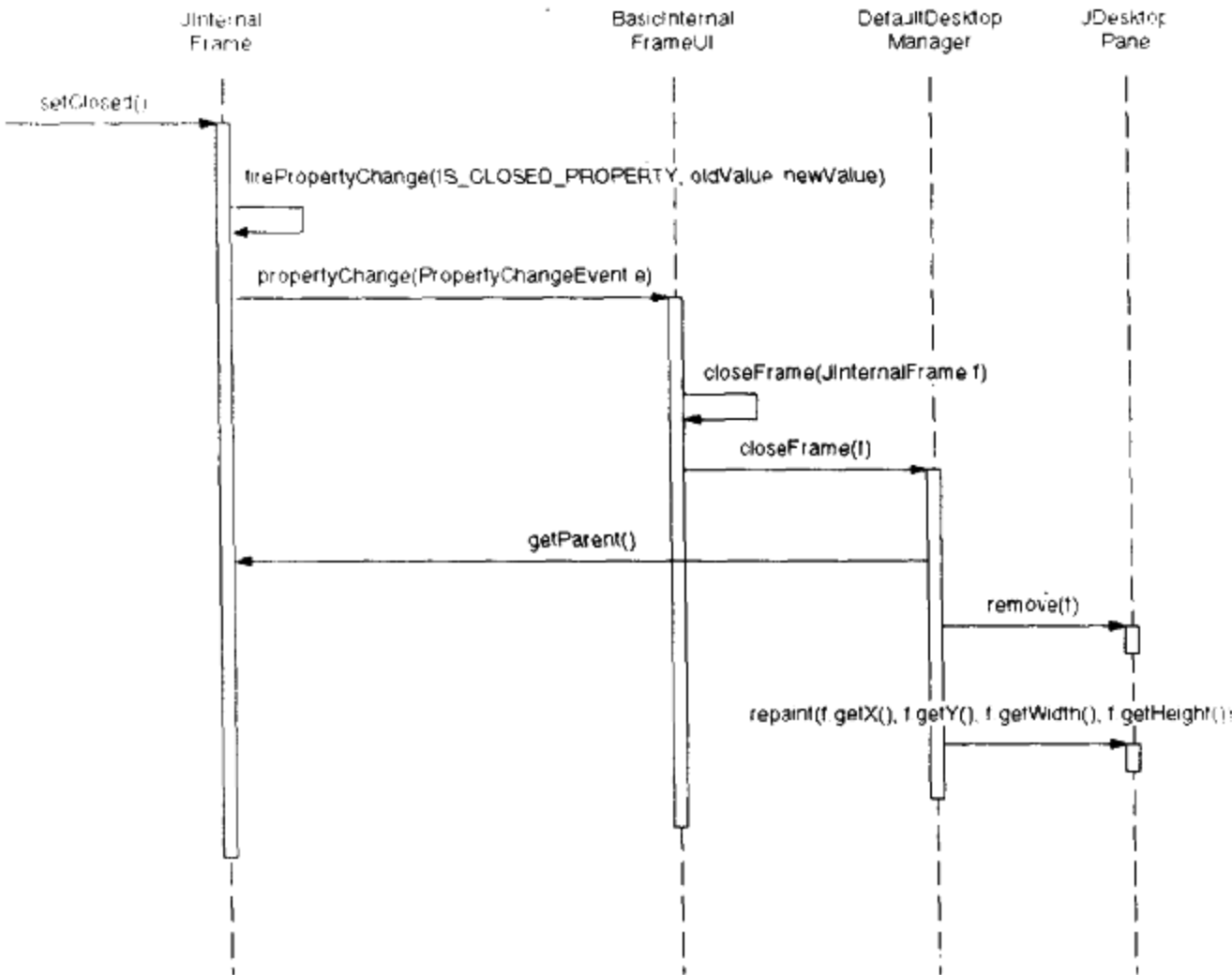


图 15-8 关闭内部窗体

其他类型的动作（这些动作可以在一个内部窗体上执行）的方案图与图 15-8 所示的图相似。调用一个 `JInternalFrame` 方法，把内部窗体交给它的 UI 代表处理，而 UI 代表又交给一个桌面管理器处理。用这种方式，特定界面样式行为被封装在一个插入式的桌面管理器中。

图 15-9 所示的小应用程序只包含一个内部窗体，这个窗体在 `JDesktopPane` 的一个实例中。这个桌面窗格配备了一个定制桌面管理器，它通过绘制这个窗体的轮廓而不是这个窗体的内容来拖动窗体和调整窗体大小^②。左边的三个图片（从上到下）说明拖动这个窗体的情况，而右边的三个图片说明调整这个窗体大小的情况。

注意 在 Swing 1.1 FCS 下，具有 `JDesktopPane.dragMode` 客户属性的桌面窗格可以通过把这个属性设置为 `outline` 来把它们的拖动模式设置为轮廓模式。如果该属性值被设置为 `null`，或者桌面窗格没有“`JDesktopPane.dragMode`”属性，则使用缺省拖动行为。

这个小应用程序创建一个 `JDesktopPaned` 实例并把这个桌面窗格的桌面管理器设置为一个 `OutlineManager` 实例。然后创建一个内部窗体并把它添加到这个桌面窗格中。

```
public class Test extends JApplet {
```

① 通常，UI 代表为了属性变化事件而监听它们的组件。
② `DefaultDesktopManager` 类拖动这个窗体的内容。

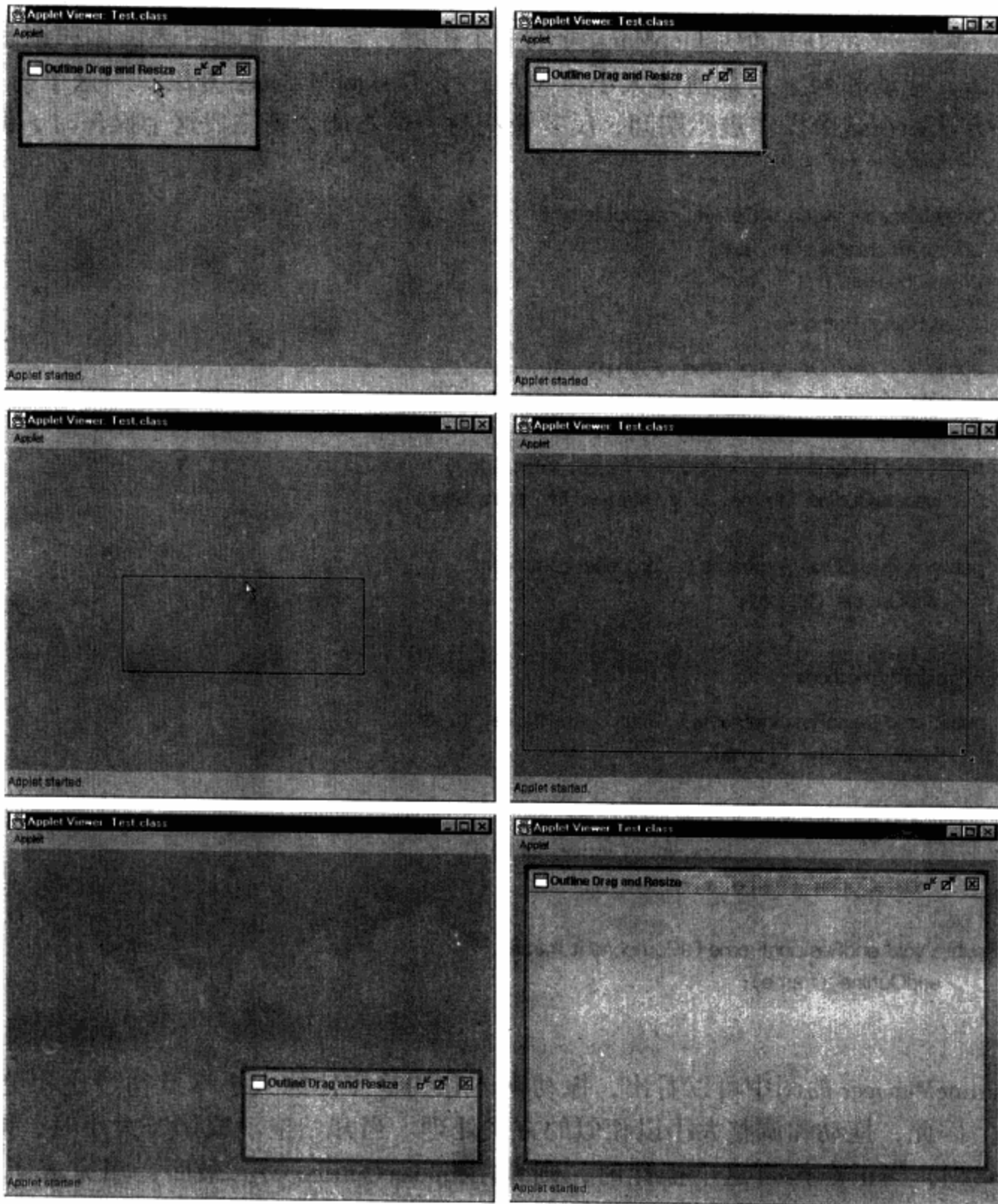


图 15-9 一个定制 DesktopManager

```

JDesktopPane desktopPane = new JDesktopPane ();

public void init () {
    Container contentPane = getContentPane ();

    contentPane.add (desktopPane, BorderLayout.CENTER);
    desktopPane.setDesktopManager (new OutlineManager ());

    JInternalFrame jif = new JInternalFrame (
        outline"Drag and ResizeMe", // title
        true, // resizable
        true, // closable
        true, // maximizable
        true); // iconifiable

    jif.setBounds (10, 10, 250, 100);
    desktopPane.add (jif);
}

```

OutlineManager 是 DefaultDesktopManager 的一个扩展，它重载拖动内部窗体和调整内部窗体大小的方法。拖动和调整内部窗体大小将调用三个 DesktopManager 方法：当这个操作开始时，调用一个方法，在这个操作进行期间，反复调用另一个方法，当完成这个操作时，调用第三个方法。

```

class OutlineManager extends DefaultDesktopManager {
    private Rectangle start, last;
    private boolean first = true;

    // dragging methods...

    public void beginDraggingFrame (JComponent frame) {
        initializeOutline (frame);
    }

    public void dragFrame (JComponent frame, int x, int y) {
        updateOutline (frame, x, y, start.width, start.height);
    }

    public void endDraggingFrame (JComponent frame) {
        endOutline (frame);
    }

    // resizing methods...

    public void beginResizingFrame (JComponent frame, int dir) {
        initializeOutline (frame);
    }

    public void resizeFrame (JComponent frame,
                             int x, int y, int w, int h) {
        updateOutline (frame, x, y, w, h);
    }

    public void endResizingFrame (JComponent frame) {
        endOutline (frame);
    }

    ...
}

```

```

Container container = frame.getParent ();
Graphics g = container.getGraphics ();
try {
    g.setXORMode (container.getBackground ());
    if (! first) {
        g.drawRect (last.x, last.y,
                    last.width-1, last.height-1);
    }
    g.drawRect (x, y, w-1, h-1);
    first = false;
}
finally {
    g.dispose ();
    last.setBounds (x, y, w, h);
}
}
...

```

`updateOutline` 方法以对这个内部窗体的一个引用和当前轮廓矩形的大小为参数。这个方法获得了对这个内部窗体的容器的引用和对这个容器的图形的引用。

把这个图形的绘制模式设置为 XOR 模式，这个模式允许绘制图形擦掉图形，而且不影响图形下面的内容。有关图形和 XOR 模式的详细内容，请参见《Java 2 图形设计卷 I：AWT》。

如果在拖动或调整大小操作开始后已经有过更新操作（即不是第一次更新操作），就在最后的更新位置上绘制一个矩形，并擦除在内部窗体的容器中绘制的最后矩形。接着，把另一个矩形绘制在新位置上。

当内部窗体的容器中的图形被更新后，清除通过调用 `getGraphics ()` 方法而获得的图形并把最后的更新位置设置为当前的位置。

当完成一个拖动或调整大小的操作时，就调用 `OutlineManager.endOutline` 方法。

```

...
private void endOutline (JComponent frame) {
    frame.setVisible (true);
    setBoundsForFrame (
        frame, last.x, last.y, last.width, last.height);
}
...

```

这个 `endOutline` 方法把这个窗体的可见性设置为 `true` 并且调用 `DefaultDesktopManager.setBoundsForFrame` 方法。

上面介绍的 `OutlineManager` 的实现有一个缺点，即直到拖动光标才能绘制窗体的轮廓。实际中，这通常没有什么问题，因为通常在按下鼠标按钮后会立即拖动光标。然而，如果在这个窗体的标题条中按下了鼠标按钮，但没有立即拖动鼠标，那么这个窗体将会被隐藏，而且看不见轮廓线。

一个较容易的修改方法是在 `initializeOutline` 方法中调用 `updateOutline ()`。如下面这样：

```

...
private void initializeOutline (final JComponent frame) {
    frame.setVisible (false);
    start = frame.getBounds ();
    last = new Rectangle (start);
}

```

```

first = true;

updateOutline (frame, start.x, start.y,
               start.width, start.height);
}
...

```

然而，上面所列的 `initializeOutline()` 的实现将不能获得所需的效果，即只有拖动鼠标才能看见窗体的轮廓线。轮廓线不可见是因为调用 `setVisible()` 方法会导致调用 `repaint()`，`repaint()` 会把一个绘制事件放在事件队列中。结果，直到对 `initializeOutline()` 的调用返回并且这个绘制事件被处理后才擦除这个窗体。调用 `updateOutline()` 会在擦除这个窗体前绘制这个窗体的轮廓线，因此这个轮廓线是不可见的。

解决办法是使用 `SwingUtilities.invokeLater` 方法：

```

...
private void initializeOutline (final JComponent frame) {

    // the call to setVisible () calls repaint, which
    // places a paint event on the event queue.
    // therefore, the effect of the setVisible () call is
    // not apparent until after this method returns

    frame.setVisible (false);
    start = frame.getBounds ();
    last = new Rectangle (start);
    first = true;

    // the Runnable below paints the initial outline
    // after the repaint event spawned by setVisible () is
    // handled

    SwingUtilities.invokeLater (new Runnable () {
        public void run () {
            updateOutline (frame, start.x, start.y,
                          start.width, start.height);
        }
    });
}
...

```

`SwingUtilities.invokeLater` 在这个绘制事件已经放在事件队列中之后，把 `Runnable` 对象也放在这个事件队列中。在处理完绘制事件和擦除了这个内部窗体后，调用 `Runnable` 的 `run` 方法且绘制这个窗体的轮廓线。有关 `SwingUtilities.invokeLater` 方法的详细内容，请参见 6.3 节“Swing 实用工具”。

例 15-6 列出了图 15-9 所示小应用程序的代码。

例 15-6 一个定制的 `DesktopManager`

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    JDesktopPane desktopPane = new JDesktopPane ();

    public void init () {
        Container contentPane = getContentPane ();
    }
}

```



```

contentPane.add (desktopPane, BorderLayout.CENTER);
desktopPane.setDesktopManager (new OutlineManager ());

JInternalFrame jif = new JInternalFrame (
    "Drag and ResizeMe", // title
    true, // resizable
    true, // closable
    true, // maximizable
    true); // iconifiable

jif.setBounds (10, 10, 250, 100);
desktopPane.add (jif);
}

class OutlineManager extends DefaultDesktopManager {
    private Rectangle start, last;
    private boolean first = true;

    // dragging ...

    public void beginDraggingFrame (JComponent frame) {
        initializeOutline (frame);
    }

    public void dragFrame (JComponent frame, int x, int y) {
        updateOutline (frame, x, y, start.width, start.height);
    }

    public void endDraggingFrame (JComponent frame) {
        endOutline (frame);
    }

    // resizing ...

    public void beginResizingFrame (JComponent frame, int dir) {
        initializeOutline (frame);
    }

    public void resizeFrame (JComponent frame,
                             int x, int y, int w, int h) {
        updateOutline (frame, x, y, w, h);
    }

    public void endResizingFrame (JComponent frame) {
        endOutline (frame);
    }

    // outline ...

    private void initializeOutline (final JComponent frame) {
        // the call to setVisible () calls repaint, which
        // places a paint event on the event queue.
        // therefore, the effect of the setVisible () call is
        // not apparent until after this method returns

        frame.setVisible (false);
        start = frame.getBounds ();
        last = new Rectangle (start);
        first = true;

        // the Runnable below paints the initial outline
        // after the repaint event spawned by setVisible () is
        // handled

        SwingUtilities.invokeLater (new Runnable () {

```

```

        public void run () {
            updateOutline (frame, start.x, start.y,
                           start.width, start.height);
        }
    }

    private void updateOutline (JComponent frame,
                               int x, int y, int w, int h) {
        Container container = frame.getParent ();
        Graphics g = container.getGraphics ();
        try {
            g.setXORMode (container.getBackground ());
            if (! first) {
                g.drawRect (last.x, last.y,
                           last.width-1, last.height-1);
            }
            g.drawRect (x, y, w-1, h-1);
            first = false;
        }
        finally {
            g.dispose ();
            last.setBounds (x, y, w, h);
        }
    }

    private void endOutline (JComponent frame) {
        frame.setVisible (true);
        setBoundsForFrame (
            frame, last.x, last.y, last.width, last.height);
    }
}

```

DesktopManager 类总结

类总结 15-4 列出了 DefaultDesktopManager 类的 public 和 protected 变量和方法。

类总结 15-4 DefaultDesktopManager

扩展: java.lang.Object

实现: DesktopManager, java.io.Serializable

1. 构造方法

public DefaultDesktopManager ()

无参数的构造方法是由编译器产生的。

2. 方法

(1) DesktopManager 方法

public void activateFrame (JInternalFrame)

public void beginDraggingFrame (JComponent)

public void beginResizingFrame (JComponent, int direction)

public void closeFrame (JInternalFrame)

public void deactivateFrame (JInternalFrame)

public void deiconifyFrame (JInternalFrame)

```
public void dragFrame (JComponent, int x, int y)
public void endDraggingFrame (JComponent)
public void endResizingFrame (JComponent)
public void iconifyFrame (JInternalFrame)
public void maximizeFrame (JInternalFrame)
public void minimizeFrame (JInternalFrame)
public void openFrame (JInternalFrame)
protected void removeIconFor (JInternalFrame)
public void setFrame (JComponent, int x, int y, int w, int h)
public void setBoundsForFrame (JComponent, int x, int y, int w, int h)
```

上面所列的这些方法都在 DesktopManager 接口中定义。当实现定制桌面管理器时，上面所列的这些方法很可能被重载。

(2) Protected 实用方法

```
protected Rectangle getBoundsForIconOf (JInternalFrame)
protected Rectangle getPreviousBounds (JInternalFrame)
protected void setPreviousBounds (JInternalFrame, Rectangle)
protected void setWasIcon (JInternalFrame, Boolean)
protected boolean wasIcon (JInternalFrame)
```

上面所列的这些 protected 方法是由 DefaultDesktopManager 类使用的实用方法

getBoundsForIconOf 方法由 DefaultDesktopManager.iconifyFrame 方法调用以确定一个内部窗体的桌面图标的边界。

当最大化、最小化或关闭一个内部窗体时，将存储这个内部窗体的范围。setPreviousBounds 和 getPreviousBounds 方法跟踪一个内部窗体的范围。

wasIcon 和 setWasIcon 方法跟踪一个内部窗体是否已经进行图标化。

15.4 本章回顾

Swing 用 JInternalFrame、JDesktopPane 和 DesktopManager 类和接口提供了实现多文档界面所需的基本类。另外，可以定制桌面管理以便加入层叠和平铺内部窗体等特性。

第 16 章 选 取 器

本章介绍两个 Swing 选取器组件，它们是：JFileChooser 和 JColorChooser，它们分别用于选取文件和颜色。

16.1 JFileChooser

文件选取器（与选项窗格一样，参见 14.3 节“JOptionPane”）是放置在对话框中的轻量组件。一旦创建了一个 JFileChooser 实例，就可以把这个实例添加到一个对话框中。而且，JFileChooser 类还提供了一些方法，这些方法把已存在的文件添加到一个模态对话框中，并且显示这个对话框。这些方法返回一个 integer 值，指出是激活了选取器的批准按钮还是清除了这个对话框。

文件选取器支持三种显示模式：只显示文件、只显示目录、和显示文件及目录。另外，文件选取器还支持单文件选取和多文件选取[○]

可以用许多不同的方法来定制文件选取器，如图 16-1 所示。图 16-1 中的上图显示调用 JFileChooser.showSaveDialog() 后显示的标准对话框。图 16-1 下中的图显示一个文件选取器，它用定制的文本作为这个对话框的标题，这个文件选取器有一个批准按钮及它的工具提示，有定制过滤器、定制图标和一个可访问组件。

特性总结 16-1 总结了文件选取器的特性。

特性总结 16-1 JFileChooser

1. 打开/保存/定制对话框

JFileChooser 提供三个方法，它们显示一个包含选取器的模态对话框，这三个方法是：showOpenDialog()、showSaveDialog()、和 showDialog()[⊖]。每个方法都设置对话框标题和在文件选取器的批准按钮上显示的字符串，并返回一个 integer 值，指出是激活了批准按钮还是取消了对话框。

2. 显示模式

JFileChooser 支持三种显示模式：只显示文件、只显示目录、和显示文本和目录。

3. 多文件选取

文件选取器可以处理单文件选取或多文件选取，但是 Swing1.1 不完全支持多文件选取。

4. 批准按钮

可以定制文件选取器批准按钮的三个方面：即批准按钮的文本、批准按钮的工具提示文本和助记键。

5. 文件过滤器

可以从文件选取器中过滤指定文件或指定文件类型。文件选取器可以有多个过滤器，但

○ Swing1.1 FCS 不完全支持多文件选取。

⊖ 直到清除了这些对话框，这些方法才返回。

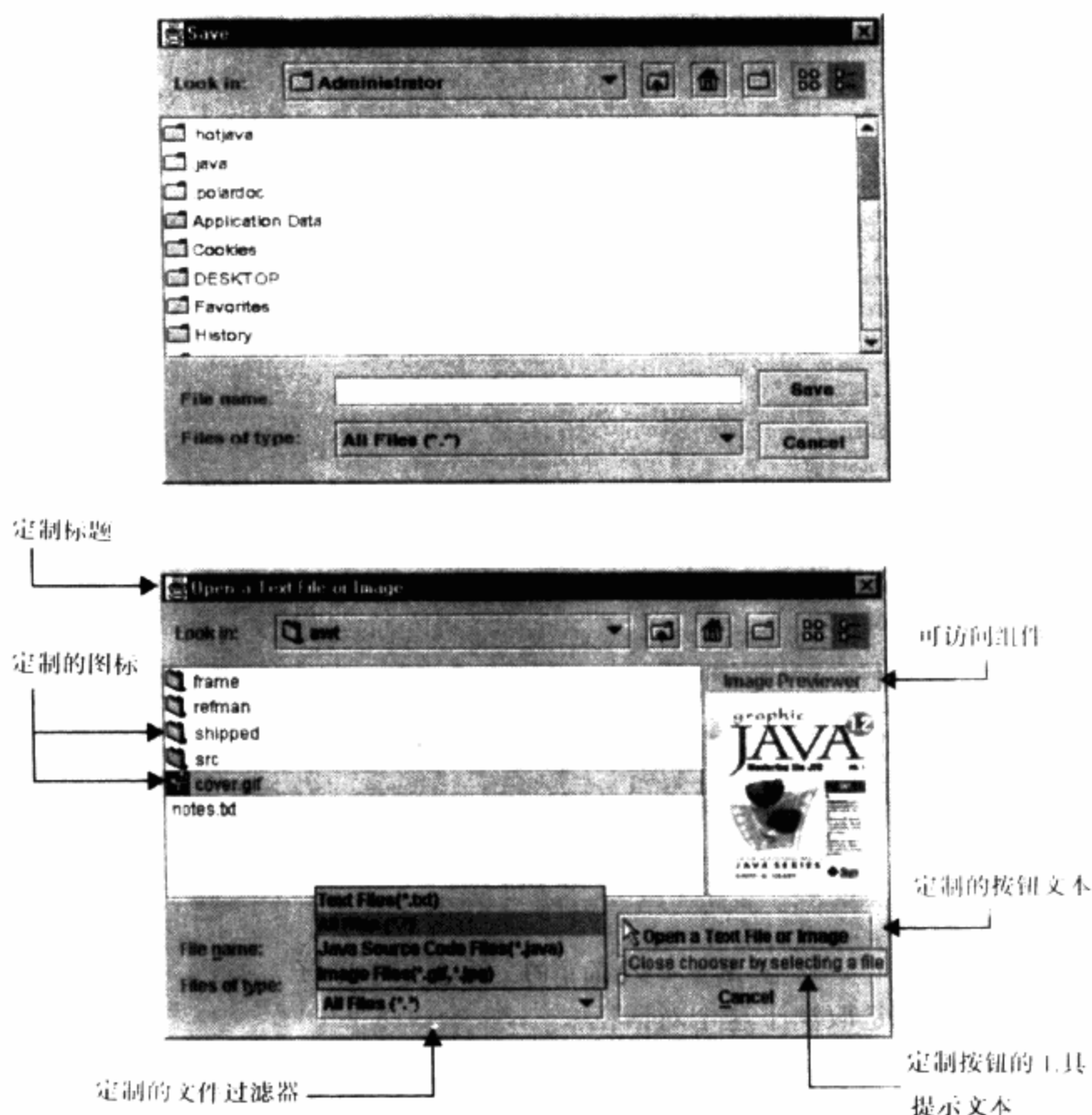


图 16-1 定制的文件选择器

是，任何给定时刻只能有一个过滤器是激活的。

6. 文件视图

给定一个文件，就可以从文件视图（它是扩展 `FileView` 类的一个对象）中获得在文件选取器中显示的文件名和图标。

7. 可访问的组件

为了许多用途，可以在一个文件选取器中安装一个组件。图像预览器等特定文件类型的预览器就是一种可访问组件。

图 16-2 示出了显示一个文件对话框的应用程序。在清除了这个对话框后，一个消息对话框将与所选取的文件进行通讯。图 16-2 的上图显示这个应用程序开始时的样子，中图显示激活这个应用程序的按钮后显示的文件选取器。下图显示在已选取文件后显示的消息对话框。

这个应用程序创建一个按钮，它被添加到这个应用程序窗体的内容窗格中。这个应用程序还用 `JFileChooser` 无参数的构造方法创建一个文件选取器。当激活这个应用程序中的按钮时，就调用 `JFileChooser.showOpenDialog()` 方法，以便显示模态对话框中的文件选取器。因为这个对话框是模态的，所以直到清除这个对话框才会返回对 `showOpenDialog()` 的调用。

在清除对话框后，`JFileChooser.getSelectedFile()` 获得对在文件选取器中选取一个文件的引用，再根据从 `JFileChooser.showOpenDialog()` 返回的值，显示一个消息对话框，指出是激活了这

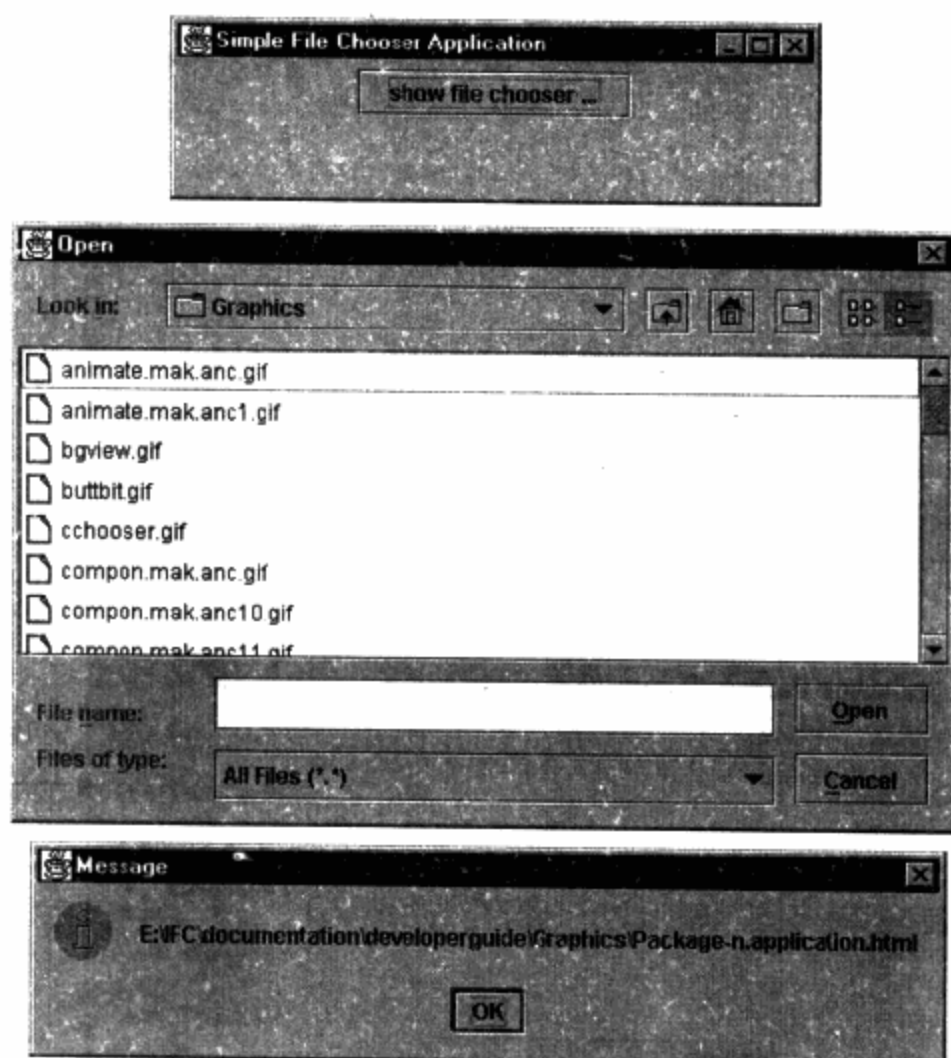


图 16-2 一个简单的文件选取器的例子

个选取器的批准按钮还是取消了这个对话框。

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    JButton button = new JButton ("show file chooser ...");

    public Test () {
        super ("Simple File Chooser Application");
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int state = chooser.showOpenDialog (null);
                File file = chooser.getSelectedFile ();

                if (file != null &&
                    state == JFileChooser.APPROVE-OPTION) {
                    JOptionPane.showMessageDialog (
                        null, file.getPath ());
                }
                else if (state == JFileChooser.CANCEL-OPTION) {
                    JOptionPane.showMessageDialog (
                        null, "Canceled");
                }
            }
        });
    }
}
```

例子 16-1 列出了图 16-2 所示应用程序的代码。

例子 16-1 一个简单的文件选取器例子

```
import java.awt.* ;
import java.awt.event.* ;
import java.io.File;
import javax.swing.* ;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    JButton button = new JButton ("show file chooser ...");

    public Test () {
        super ("Simple File Chooser Application");
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int state = chooser.showOpenDialog (null);
                File file = chooser.getSelectedFile ();

                if (file != null &&
                    state == JFileChooser.APPROVE-OPTION) {
                    JOptionPane.showMessageDialog (
                        null, file.getPath ());
                }
                else if (state == JFileChooser.CANCEL-OPTION) {
                    JOptionPane.showMessageDialog (
                        null, "Canceled");
                }
            }
        });
    }

    public static void main (String args []) {
        JFrame f = new Test ();
        f.setBounds (300, 300, 350, 100);
        f.setVisible (true);
        f.setDefaultCloseOperation (
            WindowConstants.DISPOSE-ON-CLOSE);
        f.addWindowListener (new WindowAdapter () {
            public void windowClosed (WindowEvent e) {
                System.exit (0);
            }
        });
    }
}
```


Swing 提示

检查 null 选取的文件

如果激活一个文件选取器的批准按钮，但是没有选取一个文件，则从 JFileChooser.getSelectedFile() 返回的文件将是 null。因此，在使用从 JFileChooser.getSelectedFile() 返回的文件之前，应该先进行检查，以确保实际上已经选取了一个文件。

例如，例子 16-1 所列的应用程序进行了下面的检查：

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int state = chooser.showOpenDialog(null);
        File file = chooser.getSelectedFile();

        if (file != null && state == JFileChooser.APPROVE_OPTION)
            JOptionPane.showMessageDialog(null, file.getPath());
        ...
    }
});
```

换句话说，JFileChooser.Show...() 返回 JFileChooser.APPROVE_OPTION 并不意味着选取了一个文件。

16.1.1 文件选取器类型

JFileChooser 类支持三种类型的预配置文件选取器和与它们相关联的对话框，这三种文件选取器的对话框是：定制文件选取器对话框、打开文件选取器对话框、和保存文件选取器对话框，这些对话框的区别是对话框的标题和文件选取器批准按钮上所使用的文本。

表 16-1 列出了 JFileChooser 对话框类型和用于显示这些对话框的 JFileChooser 实例方法。

表 16-1 JFileChooser 对话框类型

对话框类型	用于显示对话框的方法	对话框标题/批准按钮上的文本
定制	int showDialog (Component parent, String approveButtonText)	由第二个参数指定
打开	int showOpenDialog (Component parent)	Open
保存	int showSaveDialog (Component parent)	Save

表 16-1 所列的这些方法显示一个包含文件选取器的模态对话框，并且在清除这个对话框后返回一个 integer 值，指出是激活了这个选取器的批准按钮还是取消了这个对话框。这个对话框在传送给这些方法的父组件上居中显示。如果这些方法的父组件参数是 null，则这个对话框在屏幕的中央显示。

表 16-1 所列的方法返回下面的常量：

- JFileChooser.APPROVE_OPTION
- JFileChooser.CANCEL_OPTION

表 16-1 所列的第一个方法带一个字符串参数，这个参数是文件选取器批准按钮上的文本和对话框的标题。表 16-1 所列的后面两种方法分别把对话框的标题及文件选取器批准按钮上的文本设置为 Open 和 Save。

图 16-3 示出了一个应用程序，它用上面所列的这些方法来显示在模态对话框中的文件选

取器。这个应用程序包含一个组合框和一个显示对话框的按钮，其中，组合框显示文件选取器类型。图 16-3 中的上图显示一个保存文件选取器，中图显示一个打开文件选取器，下图显示一个定制的文件选取器，批准按钮上的文本和对话框的标题是由一个输入对话框来指定的。

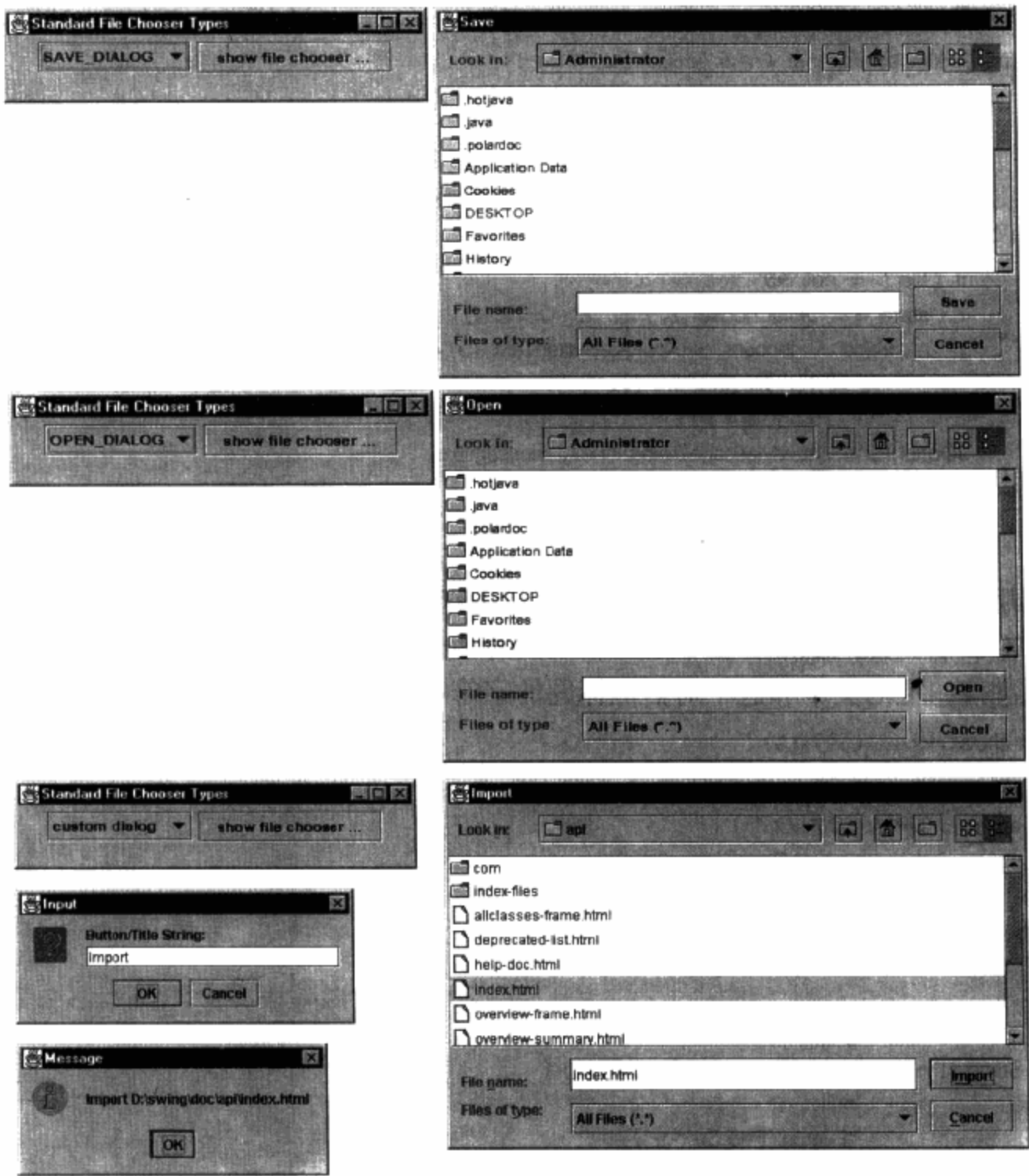


图 16-3 缺省的文件选取器类型

这个应用程序创建一个文件选取器、一个组合框和一个按钮，然后，把组合框和按钮添加到应用程序窗体的内容窗格中。当激活按钮时，根据从组合框中所选取的值，文件选取器在打开对话框、保存对话框或定制对话框中显示。当清除对话框后，将显示一个消息对话框，它指出是选取了一个文件还是取消了对话框。

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    JComboBox comboBox = new JComboBox ();
    JButton button = new JButton ("show file chooser ...");

    public Test () {
        super ("Standard File Chooser Types");
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
    }
}
```

```

contentPane.add (comboBox);
contentPane.add (button);

comboBox.addItem ("OPEN _ DIALOG");
comboBox.addItem ("SAVE _ DIALOG");
comboBox.addItem ("custom dialog");

button.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        String message = "CANCELED";
        int state = showChooser (
            (String) comboBox.getSelectedItem ());
        File file = chooser.getSelectedFile ();
        if (file != null &&
            state == JFileChooser.APPROVE _ OPTION) {
            message = chooser.getApproveButtonText () +
                " " + file.getPath ();
        }
        JOptionPane.showMessageDialog (null, message);
    }
});

```

这个应用程序的 `showChooser` 方法以从组合框中选取的字符串为参数，来显示一个相应的文件选取器对话框。如果从组合框中选取了定制对话框，则显示一个输入对话框来获得用作文件选取器批准按钮上的文本和这个对话框标题的字符串。

```

private int showChooser (String s) {
    int state;

    if (s.equals ("OPEN _ DIALOG")) {
        state = chooser.showOpenDialog (null);
    }
    else if (s.equals ("SAVE _ DIALOG")) {
        state = chooser.showSaveDialog (null);
    }
    else { // custom dialog
        String string = JOptionPane.showInputDialog (
            null,
            "Button/Title String:");

        chooser.setApproveButtonMnemonic (string.charAt (1));
        state = chooser.showDialog (Test.this, string);
    }

    return state;
}

```

例 16-2 列出了图 16-3 所示应用程序的完整代码。

例 16-2 缺省的文件选取器类型

```

import java.awt. * ;
import java.awt.event. * ;
import java.io. File;
import javax.swing. * ;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();

```

```

JComboBox comboBox = new JComboBox ();
JButton button = new JButton ("show file chooser ...");

public Test () {
    super ("Standard File Chooser Types");
    Container contentPane = getContentPane ();

    contentPane.setLayout (new FlowLayout ());
    contentPane.add (comboBox);
    contentPane.add (button);

    comboBox.addItem ("OPEN_DIALOG");
    comboBox.addItem ("SAVE_DIALOG");
    comboBox.addItem ("custom dialog");

    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            String message = "CANCELED";
            int state = showChooser (
                (String) comboBox.getSelectedItem ());
            File file = chooser.getSelectedFile ();

            if (file != null &&
                state == JFileChooser.APPROVE_OPTION) {
                message = chooser.getApproveButtonText () +
                    " " + file.getPath ();
            }

            JOptionPane.showMessageDialog (null, message);
        }
    });

    private int showChooser (String s) {
        int state;

        if (s.equals ("OPEN_DIALOG")) {
            state = chooser.showOpenDialog (null);
        }
        else if (s.equals ("SAVE_DIALOG")) {
            state = chooser.showSaveDialog (null);
        }
        else { // custom dialog
            String string = JOptionPane.showInputDialog (
                null,
                "Button/Title String:");

            chooser.setApproveButtonMnemonic (string.charAt (1));
            state = chooser.showDialog (Test.this, string);
        }

        return state;
    }

    public static void main (String args []) {
        JFrame f = new Test ();
        f.setBounds (300, 300, 350, 100);
        f.setVisible (true);

        f.setDefaultCloseOperation (
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener (new WindowAdapter () {
            public void windowClosed (WindowEvent e) {

```

```
System.exit (0);
```

```
);
```

16.1.2 可访问组件

文件选取器可以容纳一个可访问组件，如图 16-4 所示。可访问组件放在文件列表区[⊙]的右边，而且其高度与文件列表区的高度相同，其宽度与可访问组件的首选宽度相同。可访问组件有许多用途，最常见的用途是作为一个预览器，当在文件选取器中选取了文件时，预览器就显示这个文件的内容。

图 16-4 示出了一个具有图像预览器可访问组件的文件选取器。当选取了 GIF 或 JPEG 文件时，这个图像预览器将显示按比例缩小的文件图像。

图 16-4 所示的应用程序实现一个 `ImagePreviewer` 类，这个类扩展 `JLabel`。当在文件选取器中选取了一个文件时，则传送一个对所选取文件的引用给 `ImagePreviewer.configure()`。在预览器中显示的图像是以所选取文件的路径为参数创建的一个图像图标。

当创建了图像图标后，通过调用 `ImagePreviewer.getScaledInstance()` 可把图像图标重新设置为按比例缩小的图像。用这个缩小的图像来创建另一个图像图标，用它作这个标签的图标。

```
class ImagePreviewer extends JLabel {
    public void configure (File f) {
        Dimension size = getSize ();
        Insets insets = getInsets ();
        // used to create full-size image
        ImageIcon icon = new ImageIcon (f.getPath ());
        //set label's icon to new image icon
        // with scaled image
        setIcon (
            new ImageIcon (icon.getImage
                (
                    size.width-
                    insets.left-
                    insets.right,
                    size.height-in-
                    sets.top-
                    insets.bottom, Im-
                    age.SCALE_SMOOTH)));
    }
}
```

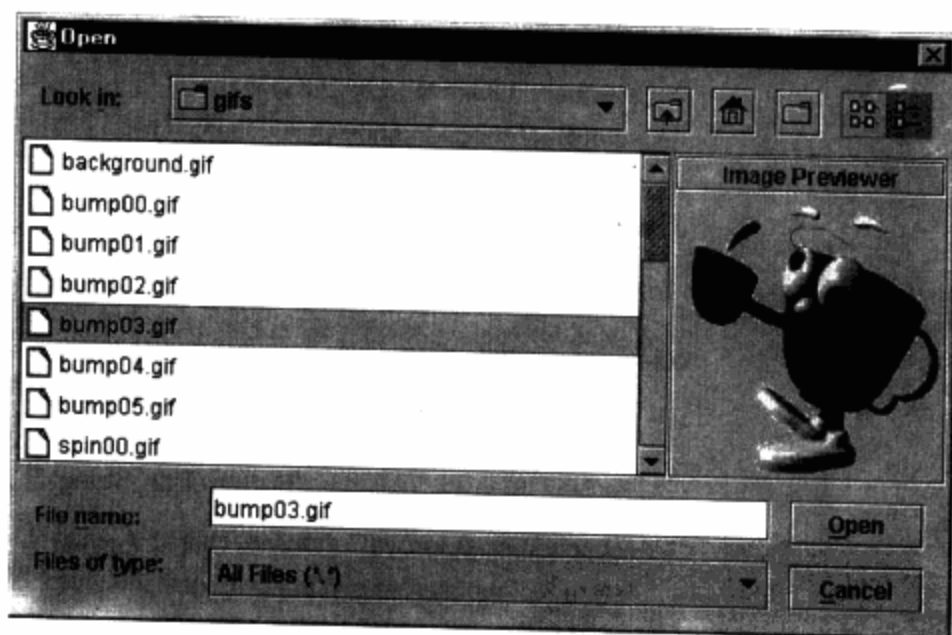


图 16-4 一个图像预览器可访问组件

⊙ 实际的位置与界面样式有关。

注意，configure 方法可以用手工方式来创建原始图像，因此不需要第二个图像图标，但是这里所呈现的实现过程更容易阅读。

把这个图像预览器放在 PreviewPanel 的一个实例中，这个实例是 JPanel 的一个扩展。PreviewPanel 类把它的首选大小设置为 (150, 0)，因为忽略了可访问组件的首选高度^①。

PreviewPanel 类还把它边框设置为蚀刻边框，并且添加这个预览器，把这个预览器作为它的中心组件。识别这个预览器的标签被指定作为预览面板的上边组件。

```
class PreviewPanel extends JPanel {
    public Previewpanel () {
        JLabel label = new JLabel ("Image Previewer",
                                   SwingConstants.CENTER);
        setPreferredSize (new Dimension (150, 0));
        setBorder (BorderFactory.createEtchedBorder ());
        setLayout (new BorderLayout ());
        label.setBorder (BorderFactory.createEtchedBorder ());
        add (label, BorderLayout.NORTH);
        add (Previewer, BorderLayout.CENTER);
    }
    ...
}
```

这个应用程序创建一个按钮、一个文件选取器、一个图像预览器和一个预览面板。这个按钮被添加到这个应用程序窗体的内容窗格中，而且通过调用 JFileChooser.setAccessory () 来把预览面板指定为文件选取器的可访问组件。

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    ImagePreviewer previewer = new ImagePreviewer ();
    PreviewPanel previewPanel = new previewPanel ();
    ...
    public Test () {
        super ("Image Previewer Accessory");
        Container contentPane = getContentPane ();
        JButton button = new JButton ("Select A File");
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);
        chooser.setAccessory (previewPanel);
        ...
    }
}
```

把调用 JFileChooser.showOpenDialog () 的动作监听器添加到这个按钮中，接着，显示一个消息对话框，它指出是选取了一个文件还是取消了这个对话框。

```
...
button.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        int state = chooser.showOpenDialog (null);
        File file = chooser.getSelectedFile ();
        String s = "CANCELED";
        if (file != null &&
```

① 结果随非标准界面样式而改变。

```

        state == JFileChooser.APPROVE_OPTION) {
            S = "File Selected:
                " + file.getPath ();
        }
        JOptionPane.showMessageDialog (null, s);
    }
}
...

```

把一个响应文件选取的属性变化监听器添加到这个文件选取器中。如果所选取的文件有 .gif 或 .jpg 的后缀，则调用 ImagePreviewer.configure () 来配置这个预览器。

```

...
chooser.addPropertyChangeListener (
    new PropertyChangeListener () {
        public void propertyChange (PropertyChangeEvent e) {
            if (e.getPropertyName ().equals (
                JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)) {
                File f = (File) e.getNewValue ();
                String s = f.getPath (), suffix = null;
                int i = s.lastIndexOf ('.')
                if (i > 0 && i < s.length () - 1)
                    suffix = s.substring (i + 1).toLowerCase ();
                if (suffix.equals ("gif") ||
                    suffix.equals ("jpg"))
                    previewer.configure (f);
            }
        }
    });
}
...

```

例 16-3 列出了图 16-4 所示应用程序的完整代码。

例 16-3 一个图像预览器可访问组件

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    ImagePreviewer previewer = new ImagePreviewer ();
    PreviewPanel previewPanel = new previewPanel ();

    class PreviewPanel extends JPanel {
        public PreviewPanel () {
            JLabel label = new JLabel ("Image Previewer",
                SwingConstants.CENTER);
            setPreferredSize (new Dimension (150, 0));
            setBorder (BorderFactory.createEtchedBorder ());
            setLayout (new BorderLayout ());
            label.setBorder (BorderFactory.createEtchedBorder ());
        }
    }
}

```



```

        add (label, BorderLayout.NORTH);
        add (Previewer, BorderLayout.CENTER);
    }
}

public Test () {
    super ("Image Previewer Accessory ");

    Container contentPane = getContentPane ();
    JButton button = new JButton ("Select A File");

    contentPane.setLayout (new FlowLayout ());
    contentPane.add (button);

    chooser.setAccessory (previewPanel);

    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            int state = chooser.showOpenDialog (null);
            File file = chooser.getSelectedFile ();
            String s = "CANCELED";
            if (file != null &&
                State == JFileChooser. APPROVE _ OPTION){
                S = "File Selected:" + file.getPath ();
            }
            JOptionPane.showMessageDialog (null, s);
        }
    });

    chooser.addPropertyChangeListener (
        new PropertyChangeListener () {
            public void propertyChange (PropertyChangeEvent e) {
                if (e.getPropertyName ().equals (
                    JFileChooser.SELECTED _ FILE _ CHANGED _ PROPERTY)) {
                    File f = (File) e.getNewValue ();
                    String s = f.getPath (), suffix = null;
                    int i = s.lastIndexOf ('.')
                    if (i > 0 && i < s.length () - 1)
                        suffix = s.substring (i + 1).toLowerCase ();

                    if (suffix.equals ("gif") ||
                        suffix.equals ("jpg")
                        previewer.configure (f);
                }
            }
        });
}

public static void main (String a []) {
    JFrame f = new Test ();
    f.setBounds (300, 300, 300, 75);
    f.setVisible (true);

    f.setDefaultCloseOperation (
        WindowConstants.DISPOSE _ ON _ CLOSE);

    f.addWindowListener (new WindowAdapter () {
        public void windowClosed (WindowEvent e) {
            System.exit (0);
        }
    });
}

```

```

|
|
class ImagePreviewer extends JLabel {
    public void configure (File f) {
        Dimension size = getSize ();
        Insets insets = getInsets ();

        ImageIcon icon = new ImageIcon (f.getPath ());

        setIcon ( new ImageIcon (icon.getImage ().getScaledInstance (
            size.width-insets.left-insets.right,
            size.height-insets.top-insets.bottom,
            Image.SCALE_SMOOTH)));
    }
}

```

16.1.3 过滤文件类型

用一个文件过滤器可以把满足某种标准的文件（如以 .gif 结尾的文件）过滤出来。任何类型的过滤器都可以与单个文件选取器相关联，但是，在给定的时刻只可以激活一个过滤器。

缺省情况下，文件选取器只使用一个 accept all（接受所有的文件）过滤器，如这个过滤器的名字所指出的，这个过滤器接受所有的文件。通过调用 JFileChooser.setFileFilter()（它替换当前的过滤器）可以用定制的过滤器来替代缺省的过滤器。

通过扩展抽象 swing.filechooser.FileFilter 类来实现文件过滤器，类总结 16-1 总结了文件过滤器。

类总结 16-1 fileFilter

扩展：java.lang.Object

1. 构造方法

public FileFilter ()

这个无参数的构造方法是由编译器产生的。

2. 方法

public abstract boolean accept (File)

public abstract String getDescription ()

accept 方法返回一个 boolean 值，它指出一个指定的文件是否被一个过滤器所接受。getDescription 方法返回一个对过滤器所接受的文件类型的简短描述。这个描述显示在文件过滤器组合框中^①。

图 16-5 所示的文件选取器有一个文本文件过滤器，这个过滤器只接受以 .txt 结尾的文件。这个文件选取器还配备了一个文本预览器，它列在例 16-4 中，但没有进行讨论。

用下面所列的应用程序来创建图 16-5 所示的文件选取器，该应用程序用 JFileChooser.setFileFilter 方法来设置选取器的文件过滤器。

```

public class Test extends JFrame {

```

① 如何显示描述取决于界面样式。

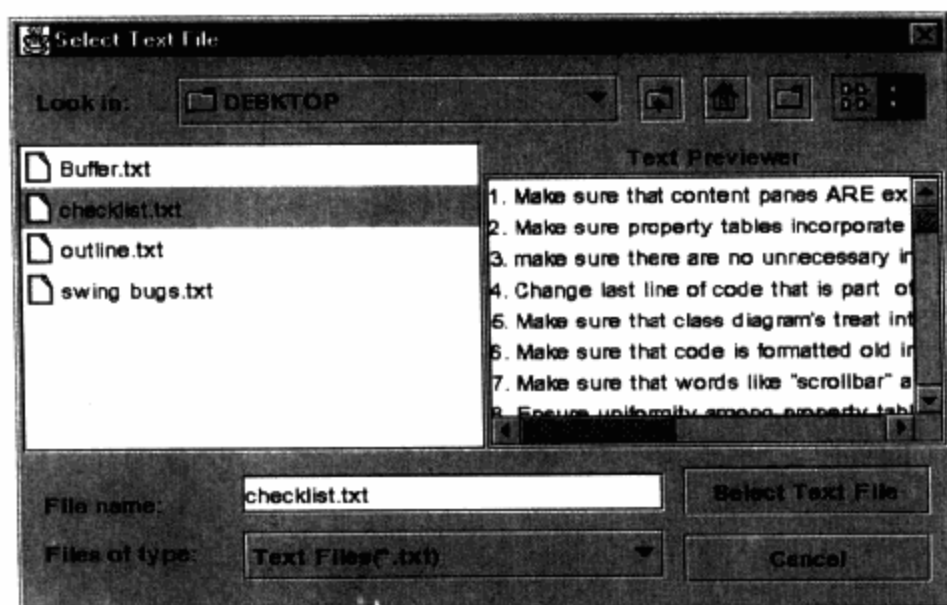


图 16-5 一个文本文件过滤器和预览器

```

JFileChooser chooser = new JFileChooser ();
TextPreviewer previewer = new TextPreviewer ();
PreviewPanel previewPanel = new PreviewPanel ();

...

public Test () {
    super ("Filtering Files");

    Container contentPane = getContentPane ();
    JButton button = new JButton ("Select A File");

    contentPane.setLayout (new FlowLayout ()); contentPane.add (button);

    chooser.setAccessory (previewPanel);
    chooser.set FileFilter (new TextFilter ());

    ...
}

```

TextFilter 类扩展 FileFilter，并且只接受以 .txt 结尾的文件。在文件选取器的文件过滤器组合框中使用了描述，如图 16-5 所示。

```

class TextFilter extends javax.swing.filechooser.FileFilter {
    public boolean accept (File f) {
        boolean accept = f.isDirectory ();

        if (! accept) {
            String suffix = getSuffix (f);

            if (suffix != null)
                accept = suffix.equals ("txt");
        }

        return accept;
    }

    public String getDescription () {
        return "Text Files (*.txt)";
    }

    private String getSuffix (File f) {
        String s = f.getPath (), suffix = null;
        int i = s.lastIndexOf ('.');

        if (i > 0 && i < s.length () - 1)

```

```

        Suffix = s.substring (i+1) .toLowerCase ();
        return suffix;
    }
}

```

例 16-4 列出了图 16-5 所示的应用程序的完整代码。

例 16-4 一个文本文件过滤器和预览器

```

import javax.swing. * ;
import java.awt. * ;
import java.awt.event. * ;
import java.beans. * ;
import java.io. * ;
import java.net.URL;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    TextPreviewer previewer = new TextPreviewer ();
    PreviewPanel previewPanel = new PreviewPanel ();

    class PreviewPanel extends JPanel {
        public PreviewPanel () {
            JLabel label = new JLabel ("Text Previewer",
                                       SwingConstants.CENTER);
            setPreferredSize (new Dimension (350, 0));
            setBorder (BorderFactory.createEtchedBorder ());
            setLayout (new BorderLayout ());

            label.setBorder (BorderFactory.createEtchedBorder ());
            add (label, BorderLayout.NORTH);
            add (previewer, BorderLayout.CENTER);
        }
    }

    public Test () {
        super ("Filtering Files");

        Container contentPane = getContentPane ();
        JButton button = new JButton ("Select A File");

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int state = chooser.showOpenDialog (null);
                File file = chooser.getSelectedFile ();
                String s = "CANCELED";

                if (file != null &&
                    state == JFileChooser.APPROVE_OPTION) {
                    s = "File Selected:" + file.getPath ();
                }

                JOptionPane.showMessageDialog (null, s);
            }
        });

        chooser.setAccessory (Preview Panel);
        chooser.setFileFilter (new TextFilter ());
    }
}

```

```

chooser.addPropertyChangeListener (
    new PropertyChangeListener () {
        public void propertyChange (PropertyChangeEvent e) {
            if (e.getPropertyName ().equals (
                JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)) {
                previewer.configure ( (File) e.getNewValue ());
            }
        }
    });
}

public static void main (String a []) {
    JFrame f = new Test ();
    f.setBounds (300, 300, 300, 75);
    f.setVisible (true);

    f.setDefaultCloseOperation (
        WindowConstants.DISPOSE_ON_CLOSE);
    f.addWindowListener (new WindowAdapter () {
        public void windowClosed (WindowEvent e) {
            System.exit (0);
        }
    });
}

class TextFilter
    extends javax.swing.filechooser.FileFilter {
    public boolean accept (File f) {
        boolean accept = f.isDirectory ();

        if (! accept) {
            String suffix = getSuffix (f);

            if (suffix != null)
                accept = suffix.equals ("txt");
        }

        return accept;
    }

    public String getDescription () {
        return "Text Files ( * .txt)";
    }

    private String getSuffix (File f) {
        String s = f.getPath (), suffix = null;
        int i = s.lastIndexOf ('.');

        if (i > 0 && i < s.length () - 1)
            suffix = s.substring (i + 1).toLowerCase ();

        return suffix;
    }
}

class TextPreviewer extends JComponent {
    private JTextArea textArea = new JTextArea ();
    private JScrollPane scrollPane = new JScrollPane (textArea);

    public TextPreviewer () {
        textArea.setEditable (false);

        setBorder (BorderFactory.createEtchedBorder ());
    }
}

```

```

        setLayout (new BorderLayout ());
        add (scrollPane, BorderLayout.CENTER);
    }
    public void configure (File file) {
        textArea.setText (contentsOfFile (file));
        SwingUtilities.invokeLater (new Runnable () {
            Public void run () {
                Jviewport vp = scrollPane.getViewport ();
                Vp.setViewPosition (new Point (0, 0));
            }
        });
    }
    public static String contentsOfFile (File file) {
        String s = new String ();
        char [] buff = new char [50000];
        InputStream is;
        InputStreamReader reader;
        URL url;
        try {
            reader = new FileReader (file);
            int nch;
            while ( (
                nch = reader.read (buff, 0, buff.length)) != -1) {
                s = s + new String (buff, 0, nch);
            }
        } catch (java.io.IOException ex) {
            s = "Could not load file";
        }
        return s;
    }
}

```

可选取的过滤器

如在上一小节中所说明的那样，`JFileChooser.setFileFilter()` 用传送给这个方法的过滤器替换缺省的文件过滤器。如果需要使用一个有多个过滤器的文件选取器，则可以使用 `JFileChooser.addChoosableFilter` 方法把一个过滤器添加到当前的过滤器列表中。可以分别使用 `JFileChooser.getChoosableFilters()` 方法和 `removeChoosableFilter()` 方法来访问当前的过滤器列表和删除过滤器。

图 16-6 示出了一个有三个过滤器的文件选取器，这三个过滤器是：文本文件过滤器、Java 源代码文件过滤器和接受所有文件的缺省过滤器。这个文件选取器还配备了一个文本预览器，它列在例 16-5 中，但没有被讨论^①。

显示图 16-6 中文件选取器的这个应用程序创建一个 `JFileChooser` 实例，并调用 `JFileChooser.addChoosableFileFilter` 来添加文本过滤器和 Java 源代码过滤器。

```
public Test () {
```

① 要了解可访问组件，请参见“可访问组件”。

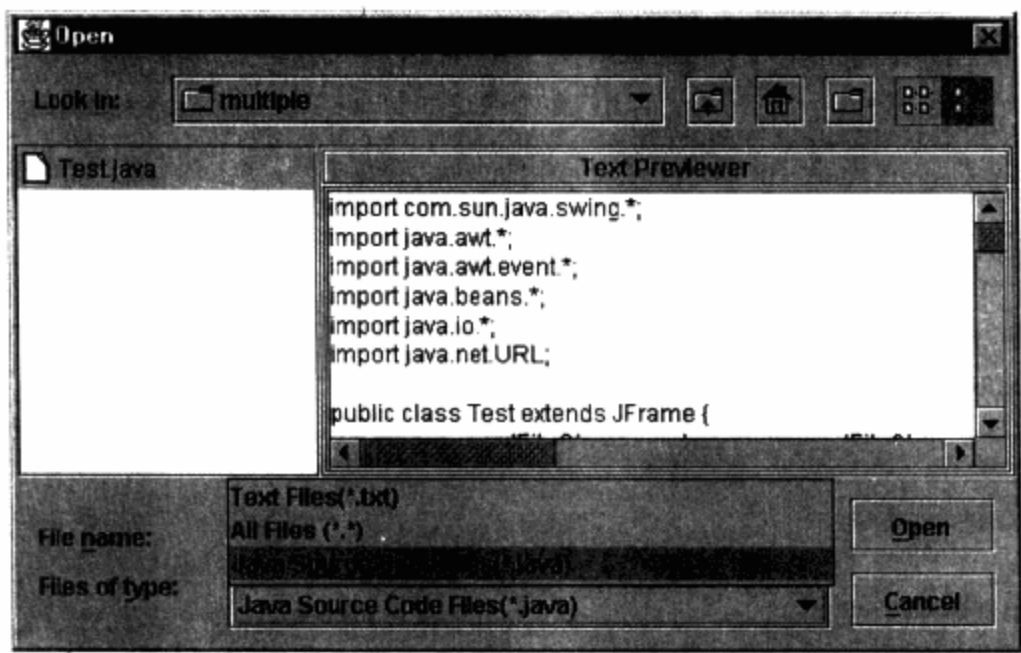


图 16-6 多种文件过滤器

```
JFileChooser chooser = new JFileChooser ();
TextPreviewer previewer = new TextPreviewer ();
Previewpanel previewPanel = new PreviewPanel ();

super ("Filtering Files");

Container contentPane = getContentPane ();
JButtonbutton = new JButton ("Select A File");

contentPane.setLayout (new FlowLayout ());
contentPane.add (button);

chooser.setAccessory (previewPanel);
chooser.addChoosableFileFilter (new TextFilter ());
chooser.addChoosableFileFilter (new JavaCodeFilter ());
...
```

因为这两个过滤器都是根据文件名后缀来确定文件的可接受性，所以这些过滤器都扩展一个认识后缀的过滤器。SuffixAwareFilter 是一个抽象过滤器，它接受所有的目录并提供一个方法，这个方法返回给定文件的文件名后缀。SuffixAwareFilter 是抽象的，因为它把 getDescription 方法留给子类去实现。

```
...
abstract class SuffixAwareFilter
    extends javax.swing.filechooser.FileFilter {
public String getSuffix (File f) {
    String s = f.getPath (), suffix = null;
    int i = s.lastIndexOf ('.');

    if (i > 0 && i < s.length () - 1)
        suffix = s.substring (i+1) .toLowerCase ();

    return suffix;
}

public boolean accept (File f) {
    return f.isDirectory ();
}
}
...
```

TextFilter 和 JavaCodeFilter 都将接受一个由 SuffixAwareFilter 超类所接受的文件（即这个文件

是一个目录), 或者分别接受文件名的后缀是 .txt 或 .java 的文件。

```
...
class JavaCodeFilter extends SuffixAwareFilter {
    public boolean accept (File f) {
        boolean accept = super.accept (f);
        if ( ! accept) {
            String suffix = getSuffix (f);
            if (suffix != null)
                accept = super.accept (f) || suffix.equals ("java");
        }
        return accept;
    }
    public String getDescription () {
        return "Java Source Code Files ( *.java)";
    }
}

class TextFilter extends SuffixAwareFilter {
    public boolean accept (File f) {
        String suffix = getSuffix (f);
        if (suffix != null)
            return super.accept (f) || suffix.equals ("txt");
        return false;
    }
    public String getDescription () {
        return "Text Files ( *.txt)";
    }
}
```

例 16-5 列出了图 16-6 所示的应用程序的完整代码。

例 16-5 多种文件过滤器

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
import java.beans.* ;
import java.io.* ;
import java.net.URL;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    TextPreviewer previewer = new TextPreviewer ();
    PreviewPanel previewPanel = new PreviewPanel ();

    class PreviewPanel extends JPanel {
        public PreviewPanel () {
            JLabel label = new JLabel ("Text Previewer",
                                      SwingConstants.CENTER);
            setPreferredSize (new Dimension (350, 0));
            setBorder (BorderFactory.createEtchedBorder ());
            setLayout (new BorderLayout ());
            label.setBorder (BorderFactory.createEtchedBorder ());
            add (label, BorderLayout.NORTH);
        }
    }
}
```

```

        add (previewer, BorderLayout.CENTER);
    }
}

public Test () {
    super ("Filtering Files");

    Container contentPane = getContentPane ();
    JButton button = new JButton ("Select A File");

    contentPane.setLayout (new FlowLayout ());
    contentPane.add (button);

    chooser.setAccessory (previewPanel);
    chooser.addChoosableFileFilter (new TextFilter ());
    chooser.addChoosableFileFilter (new JavaCodeFilter ());

    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            int state = chooser.showOpenDialog (null);
            String s = "CANCELED";

            if (state == JFileChooser.APPROVE_OPTION) {
                s = "File Selected: " +
                    chooser.getSelectedFile ().getPath ();
            }
            JOptionPane.showMessageDialog (null, s);
        }
    });

    chooser.addPropertyChangeListener (
        new PropertyChangeListener () {
            public void propertyChange (PropertyChangeEvent e) {
                if (e.getPropertyName ().equals (
                    JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)) {
                    previewer.configure ((File) e.getNewValue ());
                }
            }
        }
    );
}

public static void main (String a []) {
    JFrame f = new Test ();
    f.setBounds (300, 300, 300, 75);
    f.setVisible (true);

    f.setDefaultCloseOperation (
        WindowConstants.DISPOSE_ON_CLOSE);

    f.addWindowListener (new WindowAdapter () {
        public void windowClosed (WindowEvent e) {
            System.exit (0);
        }
    });
}

abstract class SuffixAwareFilter
    extends javax.swing.filechooser.FileFilter {
    public String getSuffix (File f) {
        String s = f.getPath (), suffix = null;
        int i = s.lastIndexOf ('.');
    }
}

```

```

        if (i > 0 && i < s.length () - 1)
            suffix = s.substring (i + 1) .toLowerCase ();
        return suffix;
    }
    public boolean accept (File f) {
        return f.isDirectory ();
    }
}

class JavaCodeFilter extends SuffixAwareFilter {
    public boolean accept (File f) {
        boolean accept = super.accept (f);
        if ( ! accept) {
            String suffix = getSuffix (f);
            if (suffix != null)
                accept = super.accept (f) || suffix.equals ("java");
        }
        return accept;
    }
    public String getDescription () {
        return "Java Source Code Files ( *.java)";
    }
}

class TextFilter extends SuffixAwareFilter {
    public boolean accept (File f) {
        String suffix = getSuffix (f);
        if (suffix != null)
            return super.accept (f) || suffix.equals ("txt");
        return false;
    }
    public String getDescription () {
        return "Text Files ( *.txt)";
    }
}

class TextPreviewer extends JComponent {
    private JTextArea textArea = new JTextArea ();
    private JScrollPane scrollPane = new JScrollPane (textArea);
    public TextPreviewer () {
        textArea.setEditable (false);
        setBorder (BorderFactory.createEtchedBorder ());
        setLayout (new BorderLayout ());
        add (scrollPane, BorderLayout.CENTER);
    }
    public void configure (File file) {
        textArea.setText (contentsOfFile (file));
        SwingUtilities.invokeLater (new Runnable () {
            public void run () {
                JViewport vp = scrollPane.getViewPort ();
                vp.setViewPosition (new Point (0, 0));
            }
        });
    }
}

```

```

static String contentsOfFile (File file) {
    String s = new String ();
    char [] buff = new char [50000];
    InputStream is;
    InputStreamReader reader;
    URL url;

    try {
        reader = new FileReader (file);

        int nch;

        while ( (
            nch = reader.read (buff, 0, buff.length)) != -1) {
            s = s + new String (buff, 0, nch);
        }

        catch (java.io.IOException ex) {
            s = "Could not load file";
        }

        return s;
    }
}

```

Swing 提示

文件过滤器

文件选取器过滤器可以被下面的 JFileChooser 方法所访问：

- void setFileFilter (swing.filechooser.FileFilter)
- void addChoosableFileFilter (swing.filechooser.FileFilter)
- void removeChoosableFileFilter (swing.filechooser.FileFilter)
- swing.filechooser.FileFilter getChoosableFileFilters ()

setFileFilter 设置当前显示的过滤器，而上面所列的其他方法对可选取的过滤器列表进行操作，这个列表在过滤器组合框中显示[⊖]。

16.1.4 文件视图

文件选取器获得图标和文件名，它们显示在一个对象中，这个对象是抽象 swing.filechooser.FileView 类的一个扩展。缺省文件视图由 BasicFileChooserUI.BasicFileView 类实现，但是，如果用 JFileChooser.setFileView 方法显式地设置一个文件视图，则这个文件视图可以完全地或部分地替代缺省文件视图。

如果文件选取器显式地配备了一个从它的任何方法中都返回 null 的文件视图，则向缺省文件视图询问信息。例如，如果文件选取器有一个文件视图（它从 getName 方法中返回 null），则从 BasicFileView 的一个实例中获得名字。因此，缺省文件视图的行为可以部分地被重载。

类总结 16-2 总结了 swing.filechooser.FileView 类。

⊖ 所使用的实际组件与界面样式有关。

类总结 16-2 FileView

扩展: `java.lang.Object`

1. 构造方法

`public FileView ()`

`FileView` 构造方法是由编译器产生的, 因为无参数构造方法是由 `FileView` 类实现的。

2. 方法

(1) 图标/名字/ 是否能展开

`public abstract Icon getIcon (File)`

`public abstract String getName (File)`

`public abstract Boolean isTraversable (File)`

上面所列的前两个方法分别返回给定文件的图标和字符串, 这个图标和字符串代表文件选取器中的文件。

`isTraversable` 方法指示一个文件 (通常一个目录) 是否可以打开。例如, 实现的一个 `FileView` 扩展 (从 `isTraversable` 中返回 `new Boolean (false)`) 将不允许打开一个目录, 在一个目录上双击将选取这个目录而不是打开这个目录。

应该指出的是, `isTraversable` 返回一个 `boolean` 对象而不是一个 `boolean` 值, 因为从 `FileView` 方法中返回 `null` 将引起与之相关联的文件选取器从缺省文件视图中得到相应的值[⊖]。

(2) 文件类型和描述

`public abstract String getDescription (File)`

`public abstract String getTypeDescription (File)`

`getDescription` 方法和 `getTypeDescription` 方法分别返回一个指定文件的描述和一个文件类型的描述。例如, 对名字为 `FileChooserExample.java` 的文件来说, `getDescription` 方法可能返回 “使用文件选取器的一个样例”, 而 `getTypeDescription` 方法可能返回 “一个 Java 源代码文件”。

对 `Swing1.1 FCS` 而言, 在 `Swing` 中不使用 `getDescription` 和 `getTypeDescription` 方法。对那些希望提供文件选取器中文件的附加信息的界面样式来说, 这些方法是有用的。

图 16-7 所示的文件选取器使用 `FileView` 类的一个定制扩展, 这个扩展为文件和目录提供了另一些图标。以 `.gif`、`.bmp`、和 `.jpg` 结尾的文件是用这样一个图标来表示的, 这个图标不同于用于其他类型文件的图标。

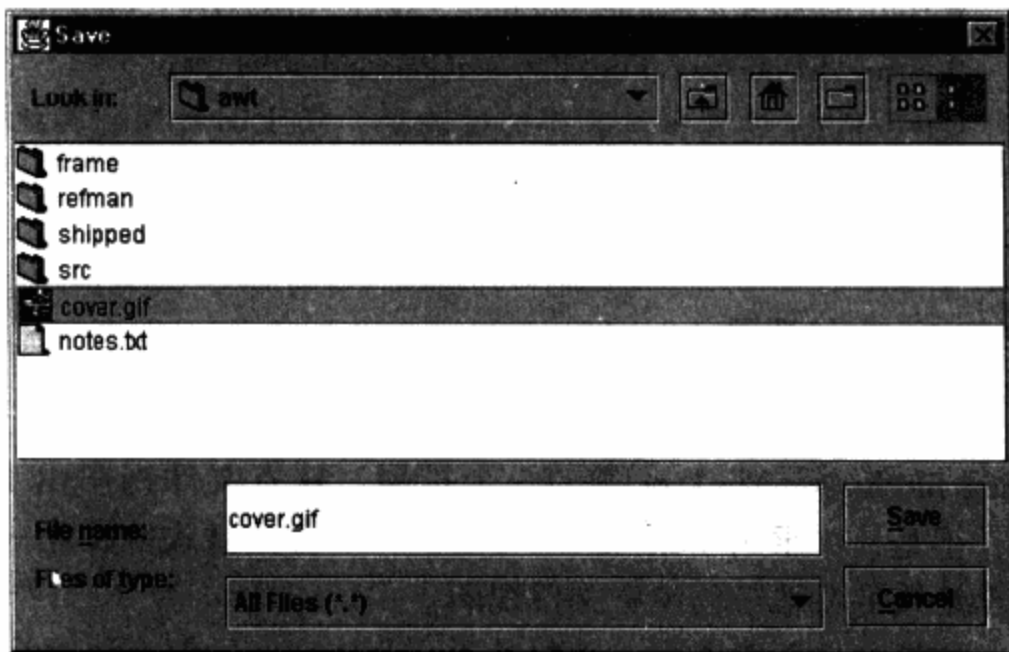


图 16-7 一个定制的文件视图

显示图 16-7 所示的文件选取器的应用程序创建 `CustomFileView` 的一个实例, 接着指定这个

⊖ `boolean` 是一个内在类型, 因此不是一个对象。

实例为这个文件选取器的文件视图。

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    JButton button = new JButton ("show file chooser ...");

    public Test () {
        super ("Custom File View Example");
        Container contentPane = getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        chooser.setFileView (new CustomFileView ());
        ...
    }
    ...
}
```

这个 CustomFileView 类实现 getName、getDescription 和 getTypeDescription 以便返回 null，这使得这些值只能从文件选取器的界面样式中获得。

CustomFileView 方法实现 getIcon 和 isTraversable 方法。CustomFileView.getIcon() 返回这种文件类型的图标，而 isTraversable 方法返回 true，除文件 (D: \ file.txt) 和目录 (D: \ books) 外。

```
class CustomFileView extends FileView {
    private Icon fileIcon = new ImageIcon ("file.gif"),
        directoryIcon = new ImageIcon ("folder.gif"),
        imageIcon = new ImageIcon ("photo.jpg");

    public String getName (File f) { return null; }
    public String getDescription (File f) { return null; }
    public String getTypeDescription (File f) { return null; }

    public Icon getIcon (File f) {
        Icon icon = null;

        if (isImage (f)) icon = imageIcon;
        else if (f.isDirectory ()) icon = directoryIcon;
        else icon = fileIcon;

        return icon;
    }

    public Boolean isTraversable (File f) {
        Boolean b;

        if (f.getPath ().equals ("D: \ \ file.txt")) {
            b = new Boolean (false);
        }
        else if (f.getPath ().equals ("D: \ \ books")) {
            b = new Boolean (false);
        }
        return b == null ? new Boolean (true) : b;
    }

    private boolean isImage (File f) {
        String suffix = getSuffix (f);
        boolean isImage = false;

        if (suffix != null) {
            isImage = suffix.equals ("gif") ||

```

```

        suffix.equals ("bmp") ||
        suffix.equals ("jpg");
    }
    return isImage;
}
private String getSuffix (File file) {
    String filestr = file.getPath (), suffix = null;
    int i = filestr.lastIndexOf ('.');
    if (i > 0 && i < filestr.length ()) {
        suffix = filestr.substring (i+1) .toLowerCase ();
    }
    return suffix;
}
}

```

例 16-6 列出了图 16-7 所示应用程序的完整代码。

例 16-6 一个定制的文件视图

```

import javax.swing. * ;
import javax.swing.filechooser.FileView;
import java.awt. * ;
import java.awt.event. * ;
import java.io. * ;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    JButton button = new JButton ("show file chooser ...");

    public Test () {
        super ("Custom File View Example");
        Container contentPane = getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        chooser.setFileView (new CustomFileView ());

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int state = chooser.showSaveDialog (null);
                File file = chooser.getSelectedFile ();
                String s = "CANCELED";

                if (state == JFileChooser.APPROVE_OPTION)
                    s = "File: " + file.getPath ();

                JOptionPane.showMessageDialog (null, s);
            }
        });
    }

    public static void main (String args []) {
        JFrame f = new Test ();
        f.setBounds (300, 300, 350, 100);
        f.setVisible (true);

        f.setDefaultCloseOperation (
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener (new WindowAdapter () {

```



```

        public void windowClosed (WindowEvent e) {
            System.exit (0);
        }
    };
}

class CustomFileView extends FileView {
    private Icon fileIcon = new ImageIcon ("file.gif"),
        directoryIcon = new ImageIcon ("folder.gif"),
        imageIcon = new ImageIcon ("photo.jpg");

    public String getName (File f) { return null; }
    public String getDescription (File f) { return null; }
    public String getTypeDescription (File f) { return null; }

    public Icon getIcon (File f) {
        Icon icon = null;

        if (isImage (f)) icon = imageIcon;
        else if (f.isDirectory ()) icon = directoryIcon;
        else icon = fileIcon;

        return icon;
    }

    public Boolean isTraversable (File f) {
        Boolean b;

        if (f.getPath ().equals ("D: \ \ file.txt")) {
            b = new Boolean (false);
        }
        else if (f.getPath ().equals ("D: \ \ books")) {
            b = new Boolean (false);
        }
        return b == null ? new Boolean (true) : b;
    }

    private boolean isImage (File f) {
        String suffix = getSuffix (f);
        boolean isImage = false;

        if (suffix != null) {
            isImage = suffix.equals ("gif") ||
                suffix.equals ("bmp") ||
                suffix.equals ("jpg");
        }

        return isImage;
    }

    private String getSuffix (File file) {
        String filestr = file.getPath (), suffix = null;
        int i = filestr.lastIndexOf ('.');

        if (i > 0 && i < filestr.length ()) {
            suffix = filestr.substring (i+1).toLowerCase ();
        }

        return suffix;
    }
}

```

16.1.5 多文件选取

JFileChooser 的实例允许选取多个文件，然而，对 Swing 1.1 FCS 而言，虽然可以在一个文

件选取器中选取多个文件，但是不能访问所选取的文件。

图 16-8 所示的文件选取器允许选取多个文件。

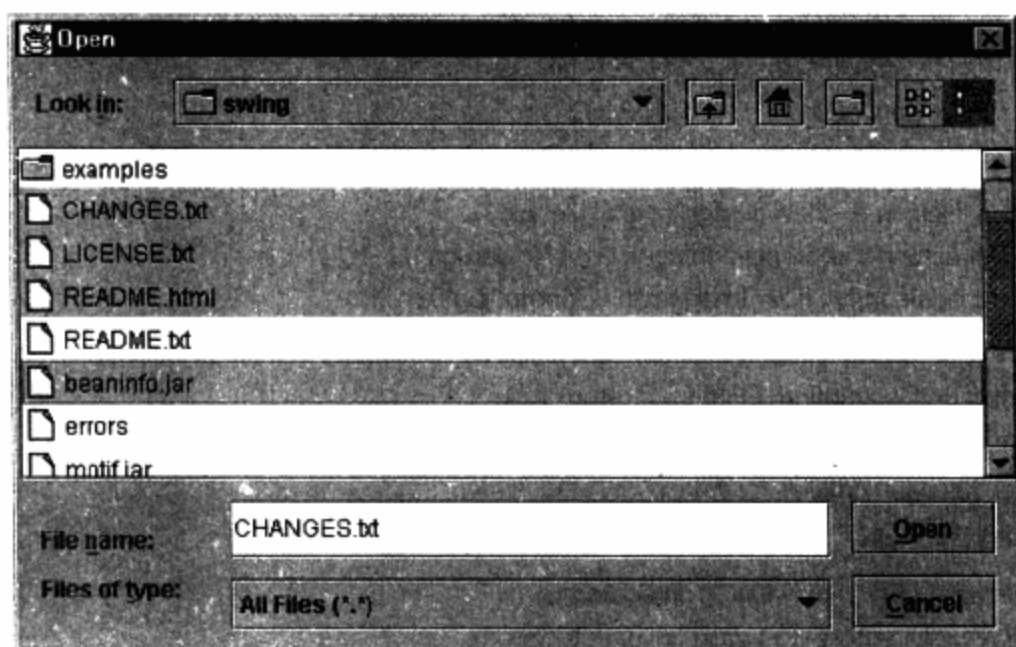


图 16-8 选取多个文件

图 16-8 所示的文件选取器调用 `JFileChooser.setMultiSelectionEnabled(true)` 来允许多文件选取。当清除了这个文件选取器后，用 `JFileChooser.getSelectedFiles()` 来获取所选取的文件，`getSelectedFiles` 方法在 Swing 1.1 FCS 下总是返回 `null`。

例 16-7 列出了图 16-8 所示的小应用程序的完整代码。

例 16-7 文件选取器的多文件选取

```
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import javax.swing.*;
import java.beans.*;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Simple File Chooser Application");
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        chooser.setMultiSelectionEnabled(true);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int state = chooser.showOpenDialog(null);
                File[] files = chooser.getSelectedFiles();
                String[] filenames = getFilenames(files);

                if (filenames != null &&
                    state == JFileChooser.APPROVE_OPTION) {
                    JOptionPane.showMessageDialog(null, filenames);
                }
            }
        });
    }
}
```

```

        else if (state == JFileChooser.CANCEL_OPTION) {
            JOptionPane.showMessageDialog (
                null, "Canceled");
        }
        else if (state == JFileChooser.ERROR_OPTION) {
            JOptionPane.showMessageDialog (
                null, "Error!");
        }
    }
}

private String [] getFilenames (File [] files) {
    String [] filenames = null;
    int numFiles = files.length;
    if (files.length > 0) {
        filenames = new String [numFiles];
        for (int i = 0; i < numFiles; ++i) {
            filenames [i] = files [i] .getPath ();
            System.out.println (filenames [i]);
        }
    }
    return filenames;
}

public static void main (String args []) {
    JFrame f = new Test ();
    f.setBounds (300, 300, 350, 100);
    f.setVisible (true);
    f.setDefaultCloseOperation (
        WindowConstants.DISPOSE_ON_CLOSE);
    f.addWindowListener (new WindowAdapter () {
        public void windowClosed (WindowEvent e) {
            System.exit (0);
        }
    });
}
}

```

组件总结 16-1 总结了 JFileChooser 类。

组件总结 16-1 JFileChooser

模型: _____

UI 代表: javax.swing.plaf.basic.BasicFileChooserUI

绘制器: _____

编辑器: _____

激发的事件: PropertyChangeEvents, ActionEvents

替换: java.awt.FileDialog

类图: 见图 16-9

JFileChooser 扩展 JComponent 并实现 Accessible 接口。JFileChooser 维护对它的可访问组件、文件过滤器和文件视图的 private 引用。JFileChooser 维护对代表 JFileChooser 属性对象的 private

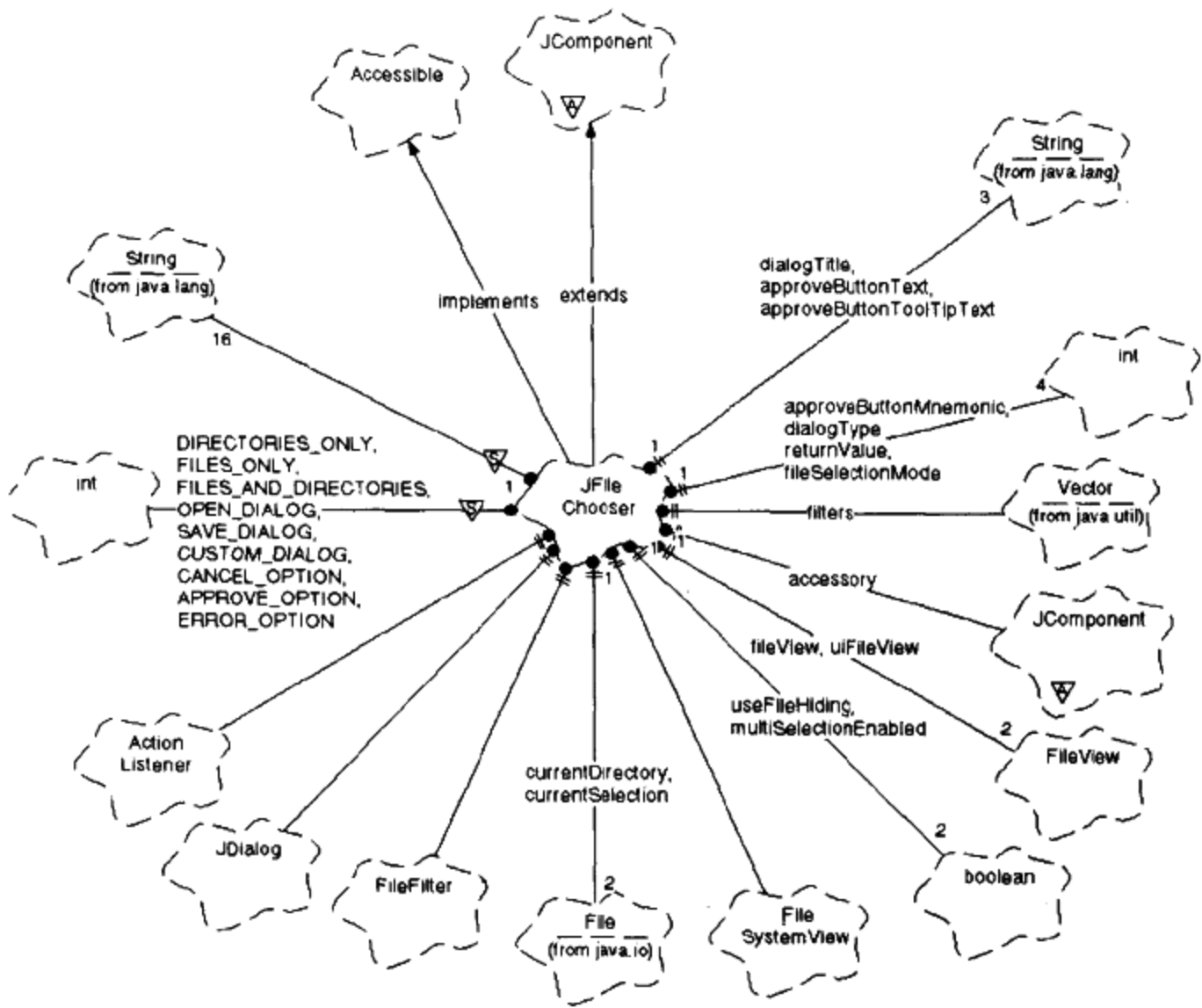


图 16-9 JFileChooser 类图

引用，JFileChooser 属性在 16.1.6 节“JFileChooser 属性”中讨论。

JFileChooser 除维护对话框类型和选取选项外，还维护一些 public static integer 值，这些值代表三个文件选取器模式：DIRECTORIES_ONLY、FILES_ONLY 和 FILES_AND_DIRECTORIES。有关由 JFileChooser 类定义的 public 字符串常量的信息请参见“JFileChooser 类总结”。

16.1.6 JFileChooser 属性

表 16-2 列出了由 JFileChooser 类维护的属性。

表 16-2 JFileChooser 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
acceptAllFileFilter	FileFilter	S	G	L&F
accessory	JComponent	B	SG	null
approveButton-Mnemonic	int	B	SG	0
approveButton-Text	String	B	SG	—
approveButton-ToolTipText	String	B	SG	null
choosableFile-Filters	FileFilter []	B	G	参见下面的介绍
currentDirectory	File	B	CSG	—
description	String	S	G	—
dialogTitle	String	B	SG	L&F

(续)

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
dialogType	int	B	SG	OPEN _ DIALOG
fileFilter		B	SG	L&F
fileHidingEnabled	boolean	B	SG	true
fileSelectionMode	int	B	SG	FILES _ ONLY
fileSystemView	FileSystemView	B	CSG	null
fileView	FileView	B	SG	null
icon	Icon	S	C	L&F
multiSelection-Enabled	boolean	B	SG	false
name	String	S	G	——
selectedFile	File	B	SG	null
selectedFiles	File []	B	SG	null
traversable	boolean	S	G	——
typeDescription	String	S	G	——

- ① B = 关联的 (激发 PropertyChangeEvent) /C = 受约束的 / I = 索引的 /
S = 简单的 /Ch = 激发 ChangeEvent
② C = 可在创建时设置 /G = 获取方法 /S = 设置方法
③ L&F = 与界面样式有关

acceptAllFileFilter——代表接受所有文件的缺省文件过滤器。这个属性由文件选取器的 UI 代表维护，使得这个属性与界面样式有关。这个属性还是只读的，而且可以用 JFileChooser.getAcceptAllFilter 方法来访问它。

有关 acceptAllFileFilter 属性和常用的文件过滤器的详细内容，请参见 16.1.3 节“过滤文件类型”。

accessory——有多种用途的 JComponent 的一个实例，最常用的可访问组件是不同文件类型的预览器。有关文件选取器可访问组件的更多信息，请参见 16.1.2 “可访问组件”。

approveButtonMnemonic——代表与文件选取器的批准按钮相关联的助记符。用 JFileChooser.setApproveButtonMnemonic 方法来设置助记符。有关常用按钮助记符的更多的信息，请参见 8.4.2 节。

approveButtonText——文件选取器配备了两个按钮，一个按钮批准所选取的文件，另一个按钮取消文件选取器对话框。与批准按钮有关的文本可以显式地设置。

通过把 dialogType 属性设置为 CUSTOM _ DIALOG 来设置批准按钮的文本。调用 JFileChooser.showOpenDialog () 或 JFileChooser.showSaveDialog () 来把批准按钮的文本设置为指定界面样式的字符串。

approveButtonToolTipText——一个文件选取器批准按钮可以有一个与之相关联的工具提示。缺省的工具提示与界面样式有关，而且可以用 JFileChooser.setApproveButtonText 方法来重载。

choosableFileFilters——一个文件选取器可以有許多与之相关联的可选择的文件过滤器。可以分别用 JFileChooser.addChoosableFileFilter 和 removeChoosableFileFilter 方法来把可选择的文件过滤器添加到一个文件选取器中或从一个文件选取器中删除。

可选择文件过滤器集总是包括 accept all 文件过滤器。有关 accept all 文件过滤器的更多信

息，请参见 `acceptAllFilter` 属性。

currentDirectory——`java.io.File` 的一个实例，它代表当前显示在文件选取器中的目录。可以在构造时或以后的任何时刻设置 `currentDirectory` 属性。

如果 `currentDirectory` 属性被指定为一个文件而不是一个目录，则当前的目录被设置为这个指定文件的第一个可到达的父目录。

dialogTitle——与界面样式有关，而且，当使用下面的 `JFileChooser` 方法时，它代表显示文件选取器的对话框的标题，这些方法是 `showOpenDialog()`、`showSaveDialog()` 和 `showDialog()`。

如果手工地把一个文件选取器放在一个对话框中，则 `dialogTitle` 属性没有影响。要了解手工地把一个文件选取器放置在一个对话框中的例子，请看例 16-8。

dialogType——可以给 `dialogType` 属性赋予下面的一个 `JFileChooser` 常量：

- `JFileChooser.OPEN_DIALOG`
- `JFileChooser.SAVE_DIALOG`
- `JFileChooser.CUSTOM_DIALOG`

如果 `dialogType` 属性被设置为 `OPEN_DIALOG` 或 `CLOSE_DIALOG`，则这个属性设置文件选取器批准按钮的文本。有关上面所列的这些常量的更多信息，请参见 `dialogType` 属性。

fileFilter——可以用两种方法中的一种来设置文件选取器的文件过滤器。这两个方法是：`JFileChooser.addChoosableFileFilter()` 或 `JFileChooser.setFileFilter()`，它们都以抽象的 `swing.filechooser.FileFilter` 类的一个扩展为参数。

`JFileChooser.setFileFilter()` 替换当前的文件过滤器，而 `addChoosableFileFilter` 则把一个过滤器添加到当前的文件过滤器列表中。

fileHidingEnabled——一个 `boolean` 值，它确定是否隐藏 `JFileChooser` 中所显示的文件。

fileSelectionMode——`fileSelectionMode` 属性可以被设置为下面值之一：

- `JFileChooser.FILES_ONLY`
- `JFileChooser.FILES_AND_DIRECTORIES`
- `JFileChooser.DIRECTORIES_ONLY`

`fileSelectionMode` 属性确定是否在文件选取器中显示文件和/（或）目录。

fileSystemView——`swing.filechooser.FileSystemView` 类的一个实例，类总结 16-3 总结了 `FileSystemView`。

类总结 16-3 `FileSystemView`

扩展：`java.lang.Object`

1. 构造方法

`public FileSystemView ()`

这个构造方法由编译器产生。

2. 方法

`public static FileSystemView getFileSystemView ()`

`public File createFileObject (File, String)`

`public File createFileObject (String)`

`public abstract File createNewFolder (File) throws IOException`

`public File [] getFiles (File, boolean)`

`public File getHomeDirectory (File)`

```

public File getParentDirectory (File)
public abstract File [] getRoots ()
public abstract boolean isHiddenFile (File)
public abstract boolean isRoot (File)

```

FileSystemView 提供指定平台的文件信息，如根分区和文件类型信息，从 JDK 1.1 java.io.File API 中不能获得这些信息。

fileView——一个对象，它的类是 abstract swing.filechooser.FileView 类的一个扩展。文件视图提供在文件选取器中显示的文件的有关信息。缺省时，由文件视图提供的信息可以从文件选取器的界面样式中获得，当然，可以显式地设置文件选取器的文件视图。

有关文件视图的详细内容，请参见 16.1.4 节“文件视图”。

multiSelectionEnabled——文件选取器支持多文件选取，multiSelectionEnabled 属性控制文件选取器是允许单文件选取还是允许多文件选取。缺省时，文件选取器允许单文件选取。

如果文件选取器只支持单文件选取，则调用 JFileChooser.getSelectedFile () 来获得所选取的文件，参见 selectedFile 属性。如果文件选取器支持多文件选取，则调用 JFileChooser.getSelectedFiles () 来获得所选取的一组文件，参见 selectedFiles 属性。

selectedFile——代表文件选取器中当前所选取的文件。可以手工从文件选取器中选取这个选取文件或用 JFileChooser.setSelectedFile 方法程序式地选取这个文件。

selectedFiles——如果文件选取器允许选取多个文件（参见 multiSelectionEnabled 属性），则可以用 JFileChooser.setSelectedFiles () 来选取这些文件。

可以用 JFileChooser.setSelectedFiles 方法程序式地设置要在文件选取器中选取的一些文件，这个方法以 java.io.File 实例的一个数组为参数。

description

name

icon

typeDescription——上面所列的这些属性实际上由文件选取器的文件视图来维护。上面所列的所有属性都是只读的，而且可以从文件选取器的文件视图获得这些值。

16.1.7 JFileChooser 事件

JFileChooser 类激发两种类型的事件：动作事件和属性变化事件。当文件选取器的批准或取消按钮被激活时，文件选取器将激发动作事件。

当修改文件选取器的关联属性时，会激发属性变化属性，要了解 JFileChooser 关联属性的列表，请参见 16.1.6 节“JFileChooser 属性”。

动作事件

缺省情况下，调用 showOpenDialog ()、showSaveDialog () 或 showDialog () 而显示的对话框都是模态的，因此，可以在调用上面所列的方法之一后，用一行代码来获得对已选取的文件（或一些文件）的引用。

// code fragment

```

int state = chooser.showOpenDialog (); //shows modal dialog
file = chooser.getSelectedFile (); //invoked after dialog
//dialog is dismissed

```


在上面所列的代码段中，代码的第二行不会被执行，直到清除了这个对话框，因此，从 `getSelectedFile()` 返回的文件代表从文件选取器选取的文件。

另一方面，非模态对话框不挂起显示对话框线程的执行。因此，不能用上述代码段中所采用的方式来获得对在非模态对话框中选取的文件的引用。例如，请看下面的代码段：

```
//code fragment

JDialog dialog = new JDialog (null, title, false);

//add chooser to dialog, set dialog title, etc.

dialog.setVisible (true);           //show non-modal dialog
file = chooser.getSelectedFile ()    //dialog has not
                                     // been dismissed
```

在上面所列的代码段中，对话框是非模态的，即调用 `JFileChooser.getSelectedFile()` 的代码行在清除对话框之前被执行。所以，从上面代码段中的 `getSelectedFile()` 返回的文件是从文件选取器选取的最后一个文件或 `null`（如果是第一次显示这个文件选取器）。

另一个响应清除文件选取器对话框的方法是向文件选取器登记一个动作监听器。当激活文件选取器的批准或取消按钮时，这个文件选取器激发一个动作事件。结果，可以用动作监听器来对从文件选取器中选取文件或取消文件选取器作出反应。

图 16-10 所示的文件选取器包含在一个非模态的对话框中。显示这个对话框的应用程序说明了对从文件选取器中选取文件作出的反应。

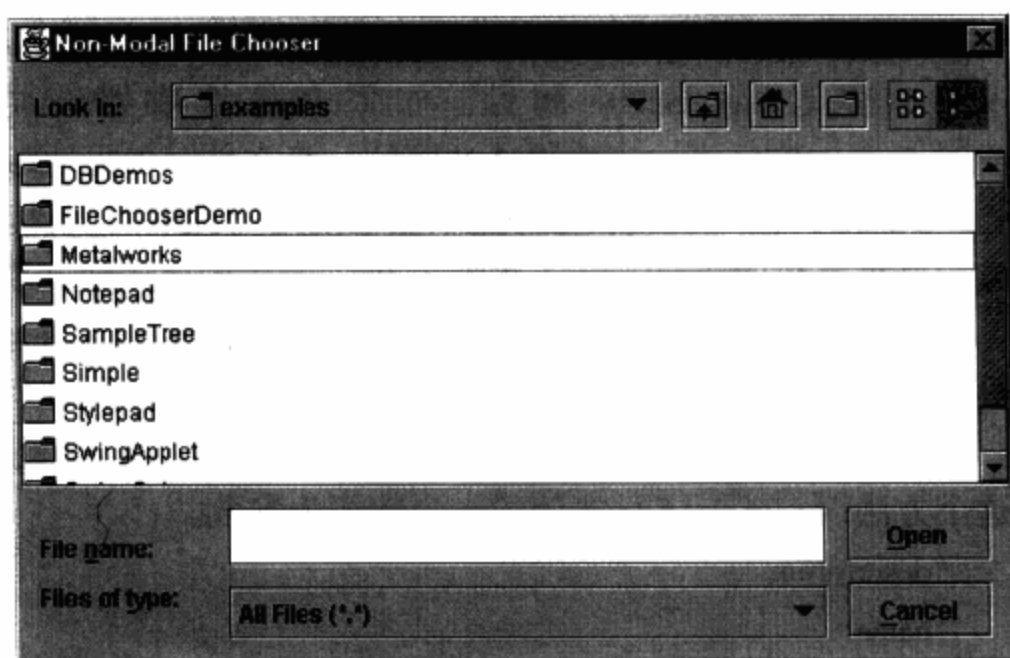


图 16-10 JFileChooser 动作事件

这个应用程序包含一个按钮，激活它将显示如图 16-10 所示的对话框。这个应用程序创建一个文件选取器和一个按钮。添加这个按钮到应用程序的内容窗格中，而且一个动作监听器将添加到这样的一个按钮中，即这个按钮创建非模态对话框、把这个文件选取器添加到这个对话框中，并且显示这个对话框。

通过第一次调用 `JFileChooser.getDialogTitle()` 来获得这个对话框的标题。如果已经显式地设置了这个对话框的标题，则调用 `JFileChooser.getDialogTitle` 将返回这个标题，否则，这个方法将返回 `null`。如果没有显式地设置这个对话框的标题，则从这个选取器的 UI 代表中获得这个标题。

以 `null` 窗体为参数来创建对话框将使对话框显示在屏幕中央。`JDialog` 构造方法的第三个参数确定对话框的模态，例如，`false` 值将显示一个非模态对话框。

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    JDialog dialog;
    JButton button = new JButton ("show file chooser ...");

    public Test () {
        super ("Simple File Chooser Application");
```

```

Container contentPane = getContentPane ();
contentPane.setLayout (new FlowLayout ());
contentPane.add (button);
button.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        String title = chooser.getDialogTitle ();
        if (title == null)
            chooser.getUI ().getDialogTitle (chooser);
        dialog = new JDialog ((Frame) null, title, false);
        Container dialogContentPane =
            dialog.getContentPane ();
        dialogContentPane.setLayout (new BorderLayout ());
        dialogContentPane.add (chooser,
            BorderLayout.CENTER);
        dialog.setTitle ("Non-Modal File Chooser");
        dialog.pack ();
        dialog.setLocationRelativeTo (Test.this);
        dialog.setVisible (true);
    }
});
...

```

把一个动作监听器添加到这个选取器本身中。这个监听器使用与动作事件有关的动作命令来确定是否从文件选取器中选取了文件或者这个对话框是否包含了已取消的选取器。如果在文件选取器中进行了一次选取，则调用 `JFileChooser.getSelectedFile` 来获取所选取的文件。然后，显示一个消息对话框，并把包含选取器对话框的对话框的可见性设置为 `false`。

```

...
chooser.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        String state = (String) e.getActionCommand ();
        File file = chooser.getSelectedFile ();

        if (file != null &&
            state.equals (JFileChooser.APPROVE_SELECTION)) {
            JOptionPane.showMessageDialog (
                null, file.getPath ());
        }
        else if (
            state.equals (JFileChooser.CANCEL_SELECTION)) {
            JOptionPane.showMessageDialog (
                null, "Canceled");
        }
        // JFileChooser action listeners are notified
        // when either the approve button or
        // cancel button is activated
        dialog.setVisible (false);
    }
});
...

```

例 16-8 列出了图 16-10 所示小应用程序的完整代码。

例 16-8 JFileChooser 动作事件

```

import java.awt. * ;
import java.awt.event. * ;
import java.io.File;
import javax.swing. * ;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser ();
    JDialog dialog;
    JButton button = new JButton ("show file chooser ...");

    public Test () {
        super ("Simple File Chooser Application");
        Container contentPane = getContentPane ();

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                String title = chooser.getDialogTitle ();

                if (title == null)
                    chooser.getUI ().getDialogTitle (chooser);

                dialog = new JDialog (null, title, false);

                Container dialogContentPane =
                    dialog.getContentPane ();

                dialogContentPane.setLayout (new BorderLayout ());
                dialogContentPane.add (chooser,
                    BorderLayout.CENTER);

                dialog.setTitle ("Non-Modal File Chooser");
                dialog.pack ();
                dialog.setLocationRelativeTo (Test.this);
                dialog.setVisible (true);
            }
        });

        chooser.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                String state = (String) e.getActionCommand ();
                File file = chooser.getSelectedFile ();

                if (file != null &&
                    state.equals (JFileChooser.APPROVE_SELECTION)) {
                    JOptionPane.showMessageDialog (
                        null, file.getPath ());
                }

                else if (
                    state.equals (JFileChooser.CANCEL_SELECTION)) {
                    JOptionPane.showMessageDialog (
                        null, "Canceled");
                }

                // JFileChooser action listeners are notified
                // when either the approve button or
                // cancel button is activated
            }
        });
    }
}

```

```

        dialog.setVisible (false);
    }
    };
}
public static void main (String args []) {
    JFrame f = new Test ();
    f.setBounds (300, 300, 350, 100);
    f.setVisible (true);
    f.setDefaultCloseOperation (
        WindowConstants.DISPOSE_ON_CLOSE);
    f.addWindowListener (new WindowAdapter () {
        public void windowClosed (WindowEvent e) {
            System.exit (0);
        }
    });
}
}

```

16.1.8 JFileChooser 类总结

类总结 16-4 列出了 JFileChooser 的 public 和 protected 变量和方法。

类总结 16-4 JFileChooser

扩展：JComponent

实现：javax.accessibility.Accessible

1. 常量

```

public static final String ACCESSORY_CHANGED_PROPERTY
public static final String APPROVE_BUTTON_MNEMONIC_CHANGED_PROPERTY
public static final String APPROVE_BUTTON_TEXT_CHANGED_PROPERTY
public static final String
    APPROVE_BUTTON_TOOL_TIP_TEXT_CHANGED_PROPERTY
public static final String CHOOSABLE_FILE_FILTER_CHANGED_PROPERTY
public static final String DIALOG_TYPE_CHANGED_PROPERTY
public static final String DIRECTORY_CHANGED_PROPERTY
public static final String FILE_FILTER_CHANGED_PROPERTY
public static final String FILE_HIDING_CHANGED_PROPERTY
public static final String FILE_SELECTION_MODE_CHANGED_PROPERTY
public static final String FILE_SYSTEM_VIEW_CHANGED_PROPERTY
public static final String FILE_VIEW_CHANGED_PROPERTY
public static final String MULTI_SELECTION_ENABLED_CHANGED_PROPERTY
public static final String SELECTED_FILE_CHANGED_PROPERTY

```

上面所列的这些常量识别由 JFileChooser 维护的属性，如下面的代码段所示：

```

// code fragment
chooser.addPropertyChangeListener (new PropertyChangeListener () {
    public void propertyChange (PropertyChangeEvent e) {

```

```

        if (e.getPropertyName().equals(
            JFileChooser.ACCESSORY_CHANGED_PROPERTY)
            // react to accessory change
        )
    }
};

```

(1) 对话框类型

```

public static final int CUSTOM_DIALOG
public static final int OPEN_DIALOG
public static final int SAVE_DIALOG

```

上面所列的这些常量通常由 JFileChooser 在内部使用，但也传送给 JFileChooser.setDialogType 方法。如果用 JFileChooser 的 showOpenDialog() 方法、showSaveDialog() 方法或 showDialog() 方法来显示文件选取器，则对话框类型分别被自动设置为 OPEN_DIALOG、SAVE_DIALOG 和 CUSTOM_DIALOG。如果不使用 JFileChooser.show... 方法来显示对话框中的文件选取器，则这个对话框类型将缺省地设置为 OPEN_DIALOG。

由于对话框类型与文件选取器批准按钮的文本相对应，所以当用 JFileChooser.show... 方法以外的方法显示文件选取器时，可能需要用 OPEN_DIALOG 或 SAVE_DIALOG 显式地设置对话框类型。(把对话框类型设置为 CUSTOM_DIALOG 不影响文件选取器批准按钮的文本。)

(2) 模式

```

public static final int DIRECTORIES_ONLY
public static final int FILES_AND_DIRECTORIES
public static final int FILES_ONLY

```

上面所列的这些常量代表 JFileChooser 类支持的三种模式。把这些常量传送给 JFileChooser.setFileSelectionMode 方法。

(3) 选项/批准-取消

```

public static final int APPROVE_OPTION
public static final int CANCEL_OPTION
public static final int ERROR_OPTION
public static final String APPROVE_SELECTION
public static final String CANCEL_SELECTION

```

JFileChooser.showDialog()、JFileChooser.showOpenDialog() 和 JFileChooser.showSaveDialog() 返回上面所列的第一组常量。如果激活了文件选取器的批准按钮，则这些方法返回 APPROVE_OPTION，如果激活了取消按钮，则这些方法返回 CANCEL_OPTION。保留 ERROR_OPTION 以便在显示文件选取器时发生错误时使用。Swing1.1 没有使用 JFileChooser.ERROR_OPTION。

当激活批准按钮或取消按钮时，文件选取器把第二组常量作为激发动作的命令字符串。

(4) 构造方法

```

public JFileChooser()
public JFileChooser(FileSystemView)
public JFileChooser(File currentDirectory)
public JFileChooser(File currentDirectory, FileSystemView)
public JFileChooser(String currentDirectoryPath)
public JFileChooser(String currentDirectoryPath, FileSystemView)
protected void setup(FileSystemView)

```

无参数的构造方法用用户的主目录作为初始显示目录来创建一个文件选取器。

传送给 JFileChooser 构造方法的文件确定初始显示的目录。如果这个文件不是一个目录，则把初始显示目录设置为这个文件可到达的第一个父目录。

用传送给 JFileChooser 构造方法的 FileSystemView 对象来获得文件信息。要了解 JFileChooser 的 fileSystemView 属性，请参见 16.1.6 节“JFileChooser 属性”。

protected setup 方法设置文件系统视图，更新 UI 代表，并更新文件选取器的 UI 代表。

2. 方法

(1) 可访问组件

```
public JComponent getAccessory ()
```

```
public void setAccessory (JComponent)
```

上面所列的这些方法是文件选取器可访问组件的访问方法。如果新的可访问组件与旧的可访问组件不同，则 setAccessory 就激发一个属性变化事件。

(2) 批准按钮

```
public int getApproveButtonMnemonic ()
```

```
public String getApproveButtonText ()
```

```
public String getApproveButtonToolTipText ()
```

```
public void setApproveButtonMnemonic (char)
```

```
public void setApproveButtonMnemonic (int)
```

```
public void setApproveButtonText (String)
```

```
public void setApproveButtonToolTipText (String)
```

可以用三种方法来定制文件选取器的批准按钮，这三种方法是：按钮的文本、按钮的工具提示和按钮的助记符。上面所列的这些方法是这个按钮属性的访问方法。

如果传送给这些方法的值与文件选取器的当前值不同，上面所列的所有设置方法都将激发属性变化事件。可以用一个 integer 值或一个 character 值来设置批准按钮的助记符。

setApproveButtonToolTipText 方法把对话框类型设置为 JFileChooser.CUSTOM_DIALOG。

(3) 布尔属性

```
public void setFileHidingEnabled (boolean)
```

```
public boolean isFileHidingEnabled ()
```

```
public boolean isDirectorySelectionEnabled ()
```

```
public boolean isFileSelectionEnabled ()
```

文件隐藏确定是否要把隐藏 (hidden) 文件 (如 UNIX 中以 ‘.’ 开头的文件) 显示在文件选取器中。

可以用上面所列的最后两个方法来确定是允许选取目录还是允许选取文件。

(4) 对话框

```
public int showDialog (Component parent, String approveButtonText)
```

```
public int showOpenDialog (Component parent)
```

```
public int showSaveDialog (Component parent)
```

```
public String getDialogTitle ()
```

```
public int getDialogType ()
```

```
public void setDialogTitle (String)
```

```
public void setDialogType (int)
```

上面所列的第一组方法显示包含一个文件选取器的模态对话框。由这些方法返回的 integer 值是下面的常量之一：

- JFileChooser.CANCEL_OPTION
- JFileChooser.APPROVE_OPTION

对话框标题和类型由上面所列的最后四个方法访问。只有用 JFileChooser.show... 方法来显示一个文件选取器时, setDialog... 方法才会发挥作用。

(5) 文件和目录

```
public void changeToParentDirectory ()
public void rescanCurrentDirectory ()

public File getCurrentDirectory ()
public void setCurrentDirectory (File)
```

changeToParentDirectory 方法把工作交给文件选取器的文件系统视图, 而 rescanCurrentDirectory 把工作交给 UI 代表。

上面所列的最后两个方法是文件选取器当前目录的访问方法。如果传送给 setCurrentDirectory 的文件不是一个目录, 则把当前目录设置为这个指定文件可到达的第一个父目录。

(6) 文件过滤器

```
public FileFilter getAcceptAllFileFilter ()
public FileFilter getFileFilter ()
public void setFileFilter (FileFilter)

public void addChoosableFileFilter (FileFilter)
public boolean removeChoosableFileFilter (FileFilter)

public FileFilter [] getChoosableFileFilter ()
public void resetChoosableFileFilter ()
```

缺省情况下, 文件选取器只有一个接受所有文件的 accept all 过滤器。可以用 getAcceptAllFileFilter 方法来访问 accept all 过滤器。可以用 getFileFilter 和 setFileFilter 来获得和设置当前选取的过滤器。

文件选取器可以有多个过滤器, 虽然在任何给定的时刻, 只可以激活一个过滤器。用 addChoosableFileFilter 可以把过滤器添加到文件选取器中, 用 removeChoosableFileFilter 方法可以从文件选取器中删除过滤器。

getChoosableFileFilter 方法返回与一个文件选取器相关联的所有过滤器, resetChoosableFileFilter 方法把列表中的过滤器重新设置为 accept all 过滤器。

(7) 文件视图

```
public FileView getFileView ()
public void setFileView (FileView)

public boolean accept (File)
public String getDescription (File)
public Icon getIcon (File)
public String getName (File)
public String getTypeDescription (File)
public boolean isTraversable (File)
```

文件选取器有一个文件视图对象, 它扩展抽象的 swing.filechooser.FileView 类, JFileChooser 提供了文件视图的访问方法。

第二组方法直接把工作交给文件选取器的文件视图。

(8) 文件系统视图

```
public FileSystemView getFileSystemView ()
public void setFileSystemView (FileSystemView)
```


文件选取器把与操作系统有关的功能交给一个对象，这个对象扩展 `abstract swing.filechooser.FileSystemView` 类。上面所列的这些方法是文件选取器的文件系统视图的访问方法。

(9) 监听器

```
public void addActionListener (ActionListener)
public void removeActionListener (ActionListener)
protected void fireActionPerformed (String)
```

当激活文件选取器的按钮时，文件选取器将激发动作事件。因此，`JFileChooser` 提供添加和删除动作监听器的方法。`protected fireActionPerformed` 方法把一个动作事件发送给所有已登记的动作监听器。

(10) 程序式的操作

```
public void approveSelection ()
public void cancelSelection ()
public void ensureFileIsVisible (File)
```

用上面所列的这些方法可以程序式地操作文件选取器。调用 `approveSelection()` 来模仿激活文件选取器的批准按钮，调用 `cancelSelection` 方法来模仿激活文件选取器的取消按钮。用户选取或取消操作与程序式地选取与取消操作没有区别。

`ensureFileIsVisible` 方法把指定的文件滚动到视图中。

注意 在 Swing 1.1 中，不是所有的界面样式都支持 `ensureFileIsVisible()`。

(11) 文件选取模式

```
public void setFileSelectionMode (int)
public int getFileSelectionMode ()
```

文件选取器支持三种选取模式，它们由下面的 `integer` 常量定义：

- `DIRECTORIES_ONLY`
- `FILES_AND_DIRECTORIES`
- `FILES_ONLY`

把这些常量传送给 `setFileSelectionMode` 方法。用 `setFileSelectionMode()` 方法可以获得当前的选取模式。

(12) 多文件选取和所选取的文件

```
public void setMultiSelectionEnabled (boolean)
public boolean isMultiSelectionEnabled ()

public File getSelectedFile ()
public void setSelectedFile (File)

public File [] getSelectedFiles ()
public void setSelectedFiles (File [])
```

文件选取器支持单文件选取和多文件选取。上面所列的第一组方法是 `boolean multiSelectionEnabled` 属性的访问方法。

上面所列的最后四个方法是所选取文件的访问方法；其中前两个方法用于具有单文件选取的文件选取器，最后的两个方法用于具有多文件选取的文件选取器。

注意 Swing 1.1 FCS 不完全支持多文件选取。

(13) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public FileChooserUI getUI ()
public String getUIClassID ()
public void updateUI ()
```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

16.1.9 AWT 兼容

除 Swing 文件选取器具有更多的功能外，AWT 的 FileDialog 和 Swing 的 JFileChooser 的主要区别是 AWT 的文件对话框处理字符串，而 JFileChooser 处理文件。

表 16-3 列出了由 java.awt.FileDialog 类实现的 public 方法和 JFileChooser 的对应方法

表 16-3 java.awt.FileDialog 方法和对应的 JFileChooser 方法

java.awt.FileDialog 的方法	JFileChooser 的对应方法
String getDirectory ()	File getCurrentDirectory ()
String getFile ()	File getSelectedFile ()
FilenameFilter getFilenameFilter ()	FileFilter getFileFilter ()
int getMode ()	int getFileSelectionMode ()
void setDirectory (String)	void setCurrentDirectory (File)
void setFile (String)	void setSelectedFile (File)
void setFilenameFilter (FilenameFilter)	void setFileFilter (FileFilter)
void setMode (int)	void setFileSelectionMode (int)

16.2 JColorChooser

由 JColorChooser 类代表的颜色选取器是允许选取一种颜色的组件。颜色选取器由两个单独的区域组成，这两个区域是：一组在标签窗格[○]中显示的颜色选取器面板和一个预览面板。预览面板显示所选取的颜色，通过替换缺省面板可以定制这两个区域。

与文件选取器和选项窗格一样，通常在一个对话框中显示颜色选取器，然而，因为颜色选取器是组件，所以它们可以包含在任何 AWT 或 Swing 容器中。

特性总结 16-2 列出了 JColorChooser 特性。

特性总结 16-2 JColorChooser

1. 颜色选取器面板

缺省时，有三个颜色选取器面板，其中的一个颜色选取器面板是当前选取的，这三个颜色选取器面板是：Swatches、HSB 和 RGB。可以删除缺省的面板，也可以添加定制的面板。

2. 预览面板

在颜色选取器上选取的颜色可以反映在预览面板中。可以用一个定制的预览面板来替代缺省的预览面板。

3. 预制对话框

○ 有多个颜色面板。

JColorChooser 提供两种 static 方法：createDialog () 和 showDialog ()，它们都创建包含颜色选取器的对话框。前者创建并返回一个对话框（这个对话框包含传送给它的颜色选取器），后者创建颜色选取器和对话框并显示这个对话框，这两种对话框都是模态的。

图 16-11 所示的小应用程序包含一个颜色选取器。颜色选取器在缺省时配备了三个选取器面板，这三个面板都在一个标签窗格中，如图 16-11 所示。

图 16-11 所示的小应用程序用 JColorChooser 无参数构造方法创建一个颜色选取器，而且这个选取器将添加到这个小应用程序的内容窗格中。

颜色选取器有一个选取模式，它是 swing.colorchooser.ColorSelectionModel 接口的一个实现。图 16-11 所示的小应用程序把一个变化监听器添加到这个颜色选取器的选取模式中，以便确定选取颜色的时间。这个监听器通过更新这个小应用程序的状态区来响应选取。

例 16-9 列出了图 16-11 所示小应用程序的代码。

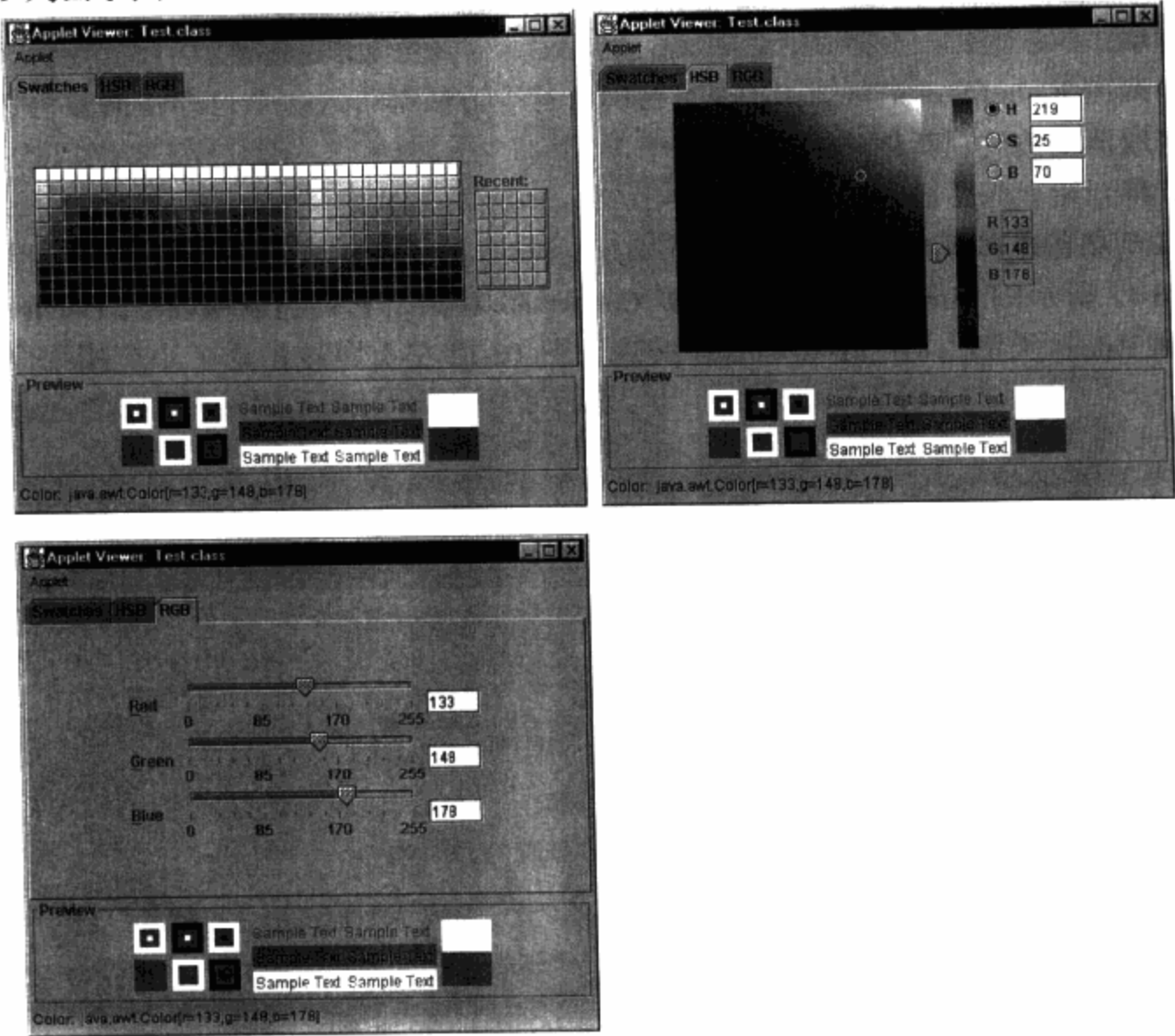


图 16-11 在一个小应用程序中显示的颜色选取器

例 16-9 在一个小应用程序中显示的颜色选取器

```
import javax.swing.*;
import javax.swing.colorchooser.*;
import javax.swing.event.*;
import java.awt.*;

public class Test extends JApplet {
    JColorChooser chooser = new JColorChooser ();
    ColorSelectionModel model = chooser.getSelectionModel ();
```

```
public void init () {
    GetContentPane ().add (chooser, BorderLayout.CENTER);

    model.addChangeListener (new ChangeListener () {
        public void stateChanged (ChangeEvent e) {
            showStatus ("Color: " + chooser.getColor ());
        }
    });
}
```

16.2.1 在对话框中显示颜色选取器

可以用两种方法中的一种方法在一个对话框中显示颜色选取器。每当调用 static JColorChooser.showDialog 方法时，这个方法都创建一个颜色选取器，并把这个颜色选取器放在新创建的对话框中。

每当调用 JColorChooser.createDialog 方法时，这个方法都带一个已存在的颜色选取器，这个颜色选取器也被放在新创建的对话框中。

下节将介绍上面所描述的这些方法。

1. 显示颜色选取器对话框

图 16-12 所示的小应用程序包含许多 ColorPatch 对象，它们包含在一个 Palette 中，ColorPatch 类和 Palette 类都由这个小应用程序实现。图 16-12 左上图显示这个小应用程序开始的样

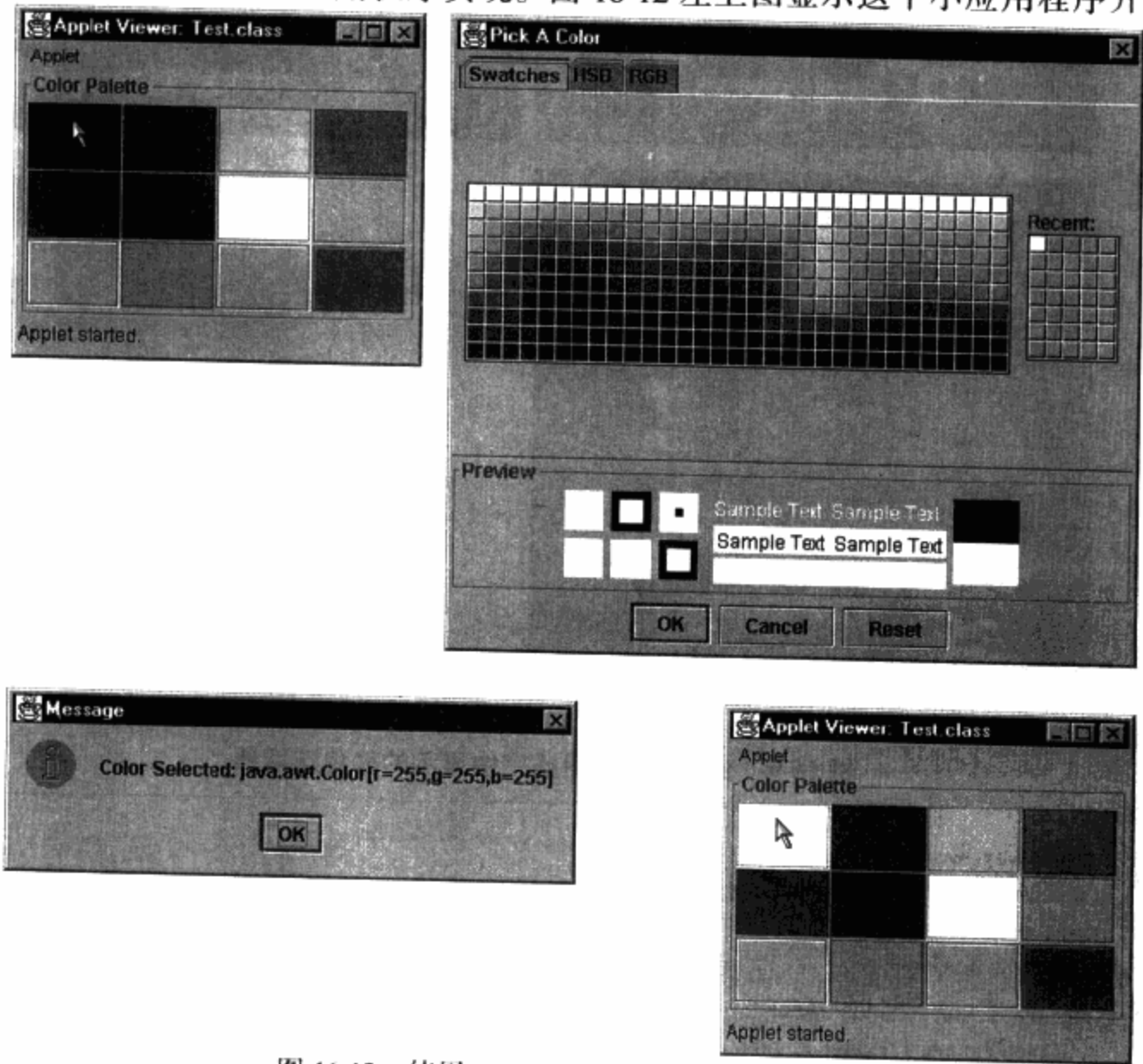


图 16-12 使用 JFileChooser.showDialog ()

子。右上图显示单击左上角颜色块后显示的颜色选取器的样子。左下图显示一个消息对话框，在清除颜色选取器后，这个小应用程序将显示这个消息对话框。右下角显示从颜色选取器中选取了白色后，小应用程序的样子。发生鼠标按下事件的那个颜色块的背景色被设置为所选取的颜色。

这个小应用程序创建 `Palette` 的一个实例，然后，把这个实例添加到这个小应用程序的内容窗格中。

```
public class Test extends JApplet {
    public void init () {
        GetContentPane ().add (new Palette (), BorderLayout.CENTER);
    }
}
```

这个 `Palette` 类是 `JPanel` 的一个简单扩展，它包含许多 `ColorPatch` 对象，这些对象代表一组缺省的颜色。

```
class Palette extends JPanel {
    private Color [] defaultColors = new Color [] {
        Color.blue, Color.red, Color.yellow, Color.green,
        Color.magenta, Color.darkGray, Color.white, Color.orange,
        Color.pink, Color.cyan, Color.lightGray, Color.gray,
    };

    public Palette () {
        int columns = 3;

        setBorder (
            BorderFactory.createTitledBorder ("Color Palette"));

        setLayout (new GridLayout (columns, 0, 1, 1));

        for (int i=0; i < defaultColors.length; ++i)
            add (new ColorPatch (defaultColors [i]));
    }
}
```

`ColorPatch` 类是这个小程序实现的最有趣的类。`ColorPatch` 还是 `JPanel` 的一个扩展，这个扩展有一个鼠标监听器，当检测到一个鼠标按下事件时，这个扩展调用 `JColorChooser.showDialog`。

`JColorChooser.showDialog` 创建一个颜色选取器和一个模态对话框、把这个选取器添加到这个对话框中、显示这个对话框、并返回对从选取器中所选取的颜色的一个引用。如果通过激活颜色选取器的取消按钮、按下 `Esc` 键、或单击这个对话框的关闭按钮，取消了这个对话框，则 `showDialog` 方法返回一个 `null` 引用。

`ColorPatch` 类的鼠标监听器显示一个消息对话框，它显示一条消息，这条消息与是选取了一个颜色还是取消了这个对话框相对应。如果选取了一个颜色，则这个监听器把激发这个对话框的颜色块的背景色设置为所选取的颜色，并重新绘制这个颜色块。

```
class ColorPatch extends JPanel {
    JApplet applet;
    Color selectedColor;

    public ColorPatch (Color color) {
        //add border and set background color...

        addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
```



```

        "Pick A Color", // dialog title
        false, // modality
        chooser,
        new OkListener (),
        new CancelListener ());

    chooser.setColor (getBackground ());
    dialog.setVisible (true);
}

};

class OkListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        Color color = chooser.getColor ();
        setBackground (color);
        repaint ();

        JOptionPane.showMessageDialog (chooser,
            "Color Selected: " + Chooser.getColor ());
    }
}

class CancelListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JOptionPane.showMessageDialog (null,
            "Color chooser Canceled");
    }
}
}

```

当与 ColorPatch 类相关联的鼠标监听器创建了对话框后，把选取器的颜色设置为这个块的背景色，而且这个对话框是可见的。

传送给 JFileChooser.createDialog 方法的动作监听器显示一个消息对话框的方式与例 16-10 所列的与 ColorPatch 类相关联的鼠标监听器显示一个消息对话框的方式相似。如果激活了颜色选取器的按钮，则用 JColorChooser.getColor 方法来获得所选取的颜色；并把所选取的颜色设置为这个颜色块的背景色，并重新绘制这个颜色块。

使用上面所列的 ColorPatch 类的这个小应用程序除 ColorPatch 类本身外，其余与例 16-10 所列的小应用程序完全相同。所以，我们没有列出这个小应用程序，但是在书后的 CD 盘中包含了这个小应用程序的代码清单。

Swing 提示

速度与方便的比较

在对话框中显示颜色选取器最容易的方法是使用 JColorChooser.showDialog 方法，然而代价却很大。每次调用 showDialog() 时，都在新创建的对话框中显示一个新创建的颜色选取器。

虽然它需要进一步改进，但手工地构造一个颜色选取器和一个对话框可以使性能大大地提高。如果在 16.2.1 节“在对话框中显示颜色选取器”和 16.2.2 节“定制颜色选取器”一节中介绍的这两个应用程序可以同时运行的话，则从鼠标在一个颜色块上单击到显示颜色选取器对话框的这段时间的差别上，可以看见这两个应用程序之间性能的差别。

16.2.2 定制颜色选取器

缺省情况下，颜色选取器有三个颜色选取器面板（都包含在一个标签窗格中）和一个预览

面板，如图 16-1 所示。虽然缺省颜色选取器面板和预览面板是最常用的颜色选取器，但是颜色选取器面板和预览面板可以用定制的版本来替换，如下面几小节中所介绍的那样。

1. 预览面板

当在颜色选取器中选取一个颜色时，缺省的预览面板的前景色设置为所选取的颜色，而且缺省的预览面板被重新绘制。

通过指定预览组件 `JColorChooser.setPreviewPanel (JComponent)` 可以定制一个颜色选取器的预览组件。然而，虽然由 `setPreviewPanel` 方法指定的组件可以替代缺省的预览面板，但是，定制预览面板在选取颜色的过程中遇到 Swing 1.1 FCS 的 `JColorChooser` 的一个错误时却不能更新。

注意 由于在前面引用的代码段中有错误，所以下面的应用程序在 Swing 1.1 FCS 下不能正常工作。它只能在修复了这个错误的程序中正常运行。

图 16-13 所示的颜色选取器包含一个定制预览面板。

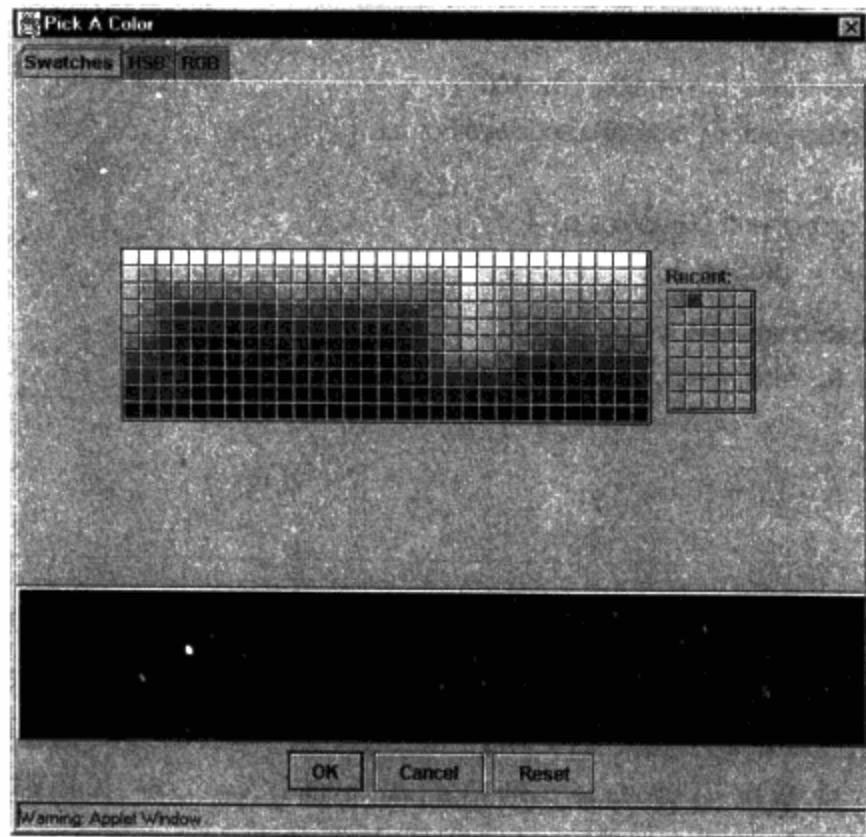


图 16-13 颜色选取器中的一个定制预览

例 16-11 列出了显示图 16-13 的颜色选取器的应用程序的代码。

例 16-11 颜色选取器中的一个定制预览

```
import javax.swing.*;
import javax.swing.colorchooser.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private JColorChooser chooser = new JColorChooser ();
    private JButton button = new JButton ("Show Color Chooser");
    private JDialog dialog;

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button, BorderLayout.CENTER);
        chooser.setPreviewPanel (new PreviewPanel ());
    }
}
```

```

button.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {

        if (dialog == null)
            dialog = JColorChooser.createDialog (
                Test.this,    // parent comp
                "Pick A Color", // dialog title
                false,        // modality
                chooser,
                null, null);

        dialog.setVisible (true);
    }
});

class PreviewPanel extends JPanel {
    public PreviewPanel () {
        setPreferredSize (new Dimension (0, 100));
        setBorder (BorderFactory.createRaisedBevelBorder ());
    }

    public void paintComponent (Graphics g) {
        Dimension size = getSize ();

        g.setColor (getForeground ());
        g.fillRect (0, 0, size.width, size.height);
    }
}

```

这个应用程序调用 `JColorChooser.setPreviewPanel()` 来设置它的颜色选取器的预览面板，`JColorChooser.setPreviewPanel()` 以 `PreviewPanel` 的一个实例为参数。

`PreviewPanel` 类扩展 `JPanel` 并指定首选的高度为 100 个像素点。颜色选取器预览面板的首选宽度被忽略，因此把首选宽度设置为 0。预览面板配备了一个突出的雕刻边框，这个边框是从 `Swing` 边框库中获得的。

当在一个颜色选取器中选取了一个颜色时，设置这个预览面板的前景色为所选取的颜色，而且把这个预览面板重新绘制。`PreviewPanel` 类重载 `paintComponent`，以使用前景色填充这个面板。遗憾的是，当选取了一个颜色时，只更新缺省的预览面板，所以，图 16-13 所示的应用程序不能正常工作。

2. 颜色选取器面板

可以用 `JColorChooser.setChooserPanels()` 来修改包含在一个颜色选取器中的一组颜色选取器面板，也可以用 `addChooserPanel` 和 `removeChooserPanel` 方法来添加或删除单个面板。本节介绍前一种方法。

在一个颜色选取器中显示的这些颜色选取器面板是一些对象，这些对象扩展 `swing.colorchooser.AbstractColorChooserPanel` 类，类总结 16-5 总结了 `AbstractColorChooserPanel` 类。

类总结 16-5 AbstractColorChooserPanel

扩展：JPanel

1. 构造方法

```
public AbstractColorChooserPanel ()
```

AbstractColorChooserPanel 类不实现任何构造方法，因此，无参数的构造方法是由编译器产生的。

2. 方法

(1) 颜色/颜色选取模式/安装选取器面板/绘制

```
public ColorSelectionMode getColorSelectionMode ()
protected Color getColorFromModel ()

public void installChooserPanel (JColorChooser)
public void uninstallChooserPanel (JColorChooser)

public void paint (Graphics)
```

上面所列的前两个方法是简便的方法，它们分别返回与这个面板的颜色选取器有关的选取模式和由选取模式维护的颜色。

当创建一个颜色选取器时，就调用 installChooserPanel 方法。这个方法调用 buildChooser() 和 updateChooser()，并把一个变化监听器添加到这个颜色选取器的选取模式中。这个监听器与 paint 方法同步，以便在绘制这个面板时调用 updateChooser() (如果在显示这个选取器之前这个选取器的选取模式发生了变化)。uninstallChooserPanel 方法删除由 installChooserPanel 方法添加到这个选取器的选取模式中的监听器。

(2) 创建和更新选取器

```
protected abstract void buildChooser ()
public abstract void updateChooser ()
```

当用 JColorChooser.setChooserPanels 方法设置一个颜色选取器的面板时，就调用 buildChooser 方法。用 buildChooser 方法来创建包含在面板中的组件并把这个组件添加到这个面板中。

当用 JColorChooser.setChooserPanels 方法设置一个颜色选取器的面板时，还调用 updateChooser 方法。而且，当选取器的颜色选取模式发生变化时，也会调用 updateChooser()。

buildChooser 方法和 updateChooser 方法是抽象的，因此必须由 AbstractColorChooserPanel 的具体扩展来实现。

(3) 显示名字和图标

```
public abstract String getDisplayName ()
public abstract Icon getLargeDisplayIcon ()
public abstract Icon getSmallDisplayIcon ()
```

上面所列的三个方法返回有关颜色选取器面板的信息。对 Swing 1.1 FCS/1.2JDK 而言，只使用了 getDisplayName 方法，用它来获取一个面板标签上所显示的字符串。例如，缺省选取器面板的缺省显示名是 Swatches、HSB、RGB，如图 16-11 所示。从 getLargeDisplayIcon() 和 getSmallDisplayIcon() 返回的图标目前在 Swing 中还没有使用。

图 16-14 所示的小应用程序包含一个按钮，激活这个按钮将在对话框中显示一个颜色选取器。颜色选取器中缺省的选取器面板由 AbstractColorChooserPanel 类的一个扩展所替代。应该注意的是，包含在图 16-14 所示的颜色选取器中的选取器面板比较简单，便于我们说明问题，但是不实用。

这个小应用程序创建一个颜色选取器和一个实现 AbstractColorChooserPanel 类的对象数组。这个数组只包含一个 ListPanel 实例，这个实例由这个小应用程序实现。注意，因为这个数组只

包含一个选取器面板，所以，标签窗格中不包含这个 ListPanel 实例。

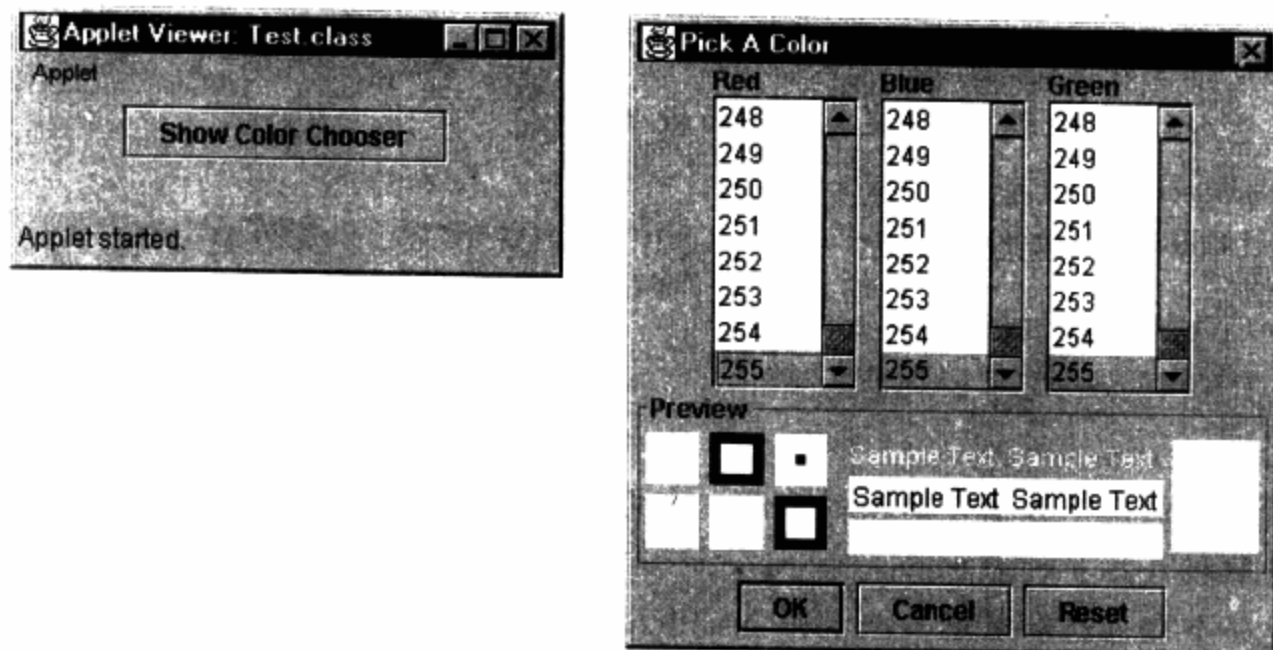


图 16-14 替换选取器面板

这个小应用程序的 `init` 方法调用 `JColorChooser.setChooserPanels()`，这个方法以颜色选取器面板数组为参数。

```
public class Test extends JApplet {
    private JColorChooser chooser = new JColorChooser ();
    private AbstractColorChooserPanel colorPanels [] =
        new AbstractColorChooserPanel [] {
            new ListPanel (),
        };
    private JButton button = new JButton ("Show Color Chooser");
    private JDialog dialog;

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button, BorderLayout.CENTER);

        chooser.setChooserPanels (colorPanels);
        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                if (dialog == null)
                    dialog = JColorChooser.createDialog (
                        Test.this, // parent comp
                        "Pick A Color", // dialog title
                        false, // modality
                        chooser,
                        null, null);
                dialog.setVisible (true);
            }
        });
    }
    ...
}
```

ListPanel 类扩展抽象的 ColorChooserPanel 类并实现 ListSelectionListener 接口。创建两个面板，

一个面板有 Red、Green、Blue 标签，另一个面板有与这些标签相关联的列表。另外，为每个列表创建了一个缺省的列表模式，并把 boolean isAdjustingValue 设置为 false。

```
...
class ListPanel extends AbstractColorChooserPanel
    implements ListSelectionListener {
    private JPanel labelPanel = new JPanel (),
        listPanel = new JPanel ();

    private JList redList = new JList (), blueList = new JList (),
        greenList = new JList ();

    private DefaultListModel redModel = new DefaultListModel (),
        blueModel = new DefaultListModel (),
        greenModel = new DefaultListModel ();

    private boolean isAdjusting = false;
    ...
```

buildChooser 方法把代表 0 到 256 值的字符串放到每个列表中。把列表添加到列表面板中，把标签添加到标签面板中，列表面板和标签面板被添加到选取器面板中。

每个列表都有指定作为列表选取监听器的选取器面板。当一个列表中的一个值发生了变化时，这个选取器面板的 valueChanged 方法就从这些列表中提取这些值，并相应地更新这个选取器的选取模式。

```
...
protected void buildChooser () {
    redList.setFixedCellWidth (50);
    greenList.setFixedCellWidth (50);
    blueList.setFixedCellWidth (50);

    for (int i=0; i < 256; ++i) {
        redModel.addElement (Integer.toString (i));
        greenModel.addElement (Integer.toString (i));
        blueModel.addElement (Integer.toString (i));
    }

    redList.setModel (redModel);
    greenList.setModel (greenModel);
    blueList.setModel (blueModel);

    listPanel.setLayout (new GridLayout (0, 3, 10, 0));

    listPanel.add (new JScrollPane (redList));
    listPanel.add (new JScrollPane (blueList));
    listPanel.add (new JScrollPane (greenList));

    labelPanel.setLayout (new GridLayout (0, 3, 10, 0));

    labelPanel.add (new JLabel ("Red"));
    labelPanel.add (new JLabel ("Blue"));
    labelPanel.add (new JLabel ("Green"));

    setLayout (new BorderLayout ());
    add (labelPanel, BorderLayout.NORTH);
    add (listPanel, BorderLayout.CENTER);

    redList.addListSelectionListener (this);
    greenList.addListSelectionListener (this);
    blueList.addListSelectionListener (this);
}
```

```

public void valueChanged (ListSelectionEvent e) {
    int r = redList.getSelectedIndex (),
        b = blueList.getSelectedIndex (),
        g = greenList.getSelectedIndex ();

    if (r != -1 && g != -1 && b != -1)
        getColorSelectionModel ().setSelectedColor (
            new Color (r, g, b));
}

```

updateChooser 方法根据选取器的选取模式的当前颜色来更新这些列表中所选取的值。

为了避免调用 JList.setSelectedIndex () 时所造成的死循环, 则使用 isAdjustingboolean 变量, 下面列举了各个细节。

调用 JList.setSelectedIndex () 引起列表激发一个列表选取事件, 导致对 ListPanel.valueChange () 的调用 (前面已说过, ListPanel 把自己作为列表的监听器添加到每个列表中)。valueChange () 更新选取器的选取模式, 引起调用 ListPanel.updateChooser (), 而 updateChooser 又调用 JList.setSelectedIndex (),

```

...
public void updateChooser () {
    if (! isAdjusting) {
        isAdjusting = true;

        Color color = getColorFromModel ();
        int r = color.getRed (), g = color.getGreen (),
            b = color.getBlue ();

        redList.setSelectedIndex (r);
        redList.ensureIndexIsVisible (r);

        blueList.setSelectedIndex (b);
        blueList.ensureIndexIsVisible (b);

        greenList.setSelectedIndex (g);
        greenList.ensureIndexIsVisible (g);

        isAdjusting = false;
    }
}
...

```

getDisplayname、getSmallDisplayIcon 和 getLargeDisplayIcon 方法必须由 ListPanel 类来实现, 因为这些方法是 AbstractColorChooserPanel 中的抽象方法。由于当前的 Swing 中没有使用图标, 所以 get...Icon 方法将返回 null 引用。getDisplayname 方法返回一个非 null 字符串, 通常把这个字符串作为与这个选取面板相关联的标签中显示的字符串。然而, 因为 ListPanel 实例是颜色选取器中显示的唯一一个选取器面板, 所以从 getDisplayName 返回的字符串使用没有。

```

...
public String getDisplayName () {
    return "lists";
}

public Icon getSmallDisplayIcon () {
    return null;
}

public Icon getLargeDisplayIcon () {
    return null;
}

```


例 16-12 列出了图 16-14 所示的小应用程序的完整代码。

例 16-12 实现一个定制颜色选取器面板

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.colorchooser.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    private JColorChooser chooser = new JColorChooser ();
    private AbstractColorChooserPanel colorPanels [] =
        new AbstractColorChooserPanel [] {
            new ListPanel (),
        };
    private JButton button = new JButton ("Show Color Chooser");
    private JDialog dialog;

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button, BorderLayout.CENTER);
        chooser.setChooserPanels (colorPanels);
        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                if (dialog == null)
                    dialog = JColorChooser.createDialog (
                        Test.this, // parent comp
                        "Pick A Color", // dialog title
                        false, // modality
                        chooser,
                        null, null);
                dialog.setVisible (true);
            }
        });
    }

    class ListPanel extends AbstractColorChooserPanel
        implements ListSelectionListener {
        private JPanel labelPanel = new JPanel (),
            listPanel = new JPanel ();

        private JList redList = new JList (), blueList = new JList (),
            greenList = new JList ();

        private DefaultListModel redModel = new DefaultListModel (),
            blueModel = new DefaultListModel (),
            greenModel = new DefaultListModel ();

        private boolean isAdjusting = false;

        public void updateChooser () {
            if (! isAdjusting) {
```

```

        isAdjusting = true;

        Color color = getColorFromModel ();
        int r = color.getRed (); g = color.getGreen ();
        b = color.getBlue ();

        redList.setSelectedIndex (r);
        redList.ensureIndexIsVisible (r);

        blueList.setSelectedIndex (b);
        blueList.ensureIndexIsVisible (b);

        greenList.setSelectedIndex (g);
        greenList.ensureIndexIsVisible (g);

        isAdjusting = false;
    }

    protected void buildChooser () {
        redList.setFixedCellwidth (50);
        greenList.setFixedCellwidth (50);
        blueList.setFixedCellwidth (50);

        for (int i=0; i < 256; ++i) {
            redModel.addElement (Integer.toString (i));
            greenModel.addElement (Integer.toString (i));
            blueModel.addElement (Integer.toString (i));
        }

        redList.setModel (redModel);
        greenList.setModel (greenModel);
        blueList.setModel (blueModel);

        listPanel.setLayout (new GridLayout (0, 3, 10, 0));

        listPanel.add (new JScrollPane (redList));
        listPanel.add (new JScrollPane (blueList));
        listPanel.add (new JScrollPane (greenList));

        labelPanel.setLayout (new GridLayout (0, 3, 10, 0));

        labelPanel.add (new JLabel ("Red"));
        labelPanel.add (new JLabel ("Blue"));
        labelPanel.add (new JLabel ("Green"));

        setLayout (new BorderLayout ());
        add (labelPanel, BorderLayout.NORTH);
        add (listPanel, BorderLayout.CENTER);

        redList.addListSelectionListener (this);
        greenList.addListSelectionListener (this);
        blueList.addListSelectionListener (this);
    }

    public void valueChanged (ListSelectionEvent e) {
        int r = redList.getSelectedIndex ();
        b = blueList.getSelectedIndex ();
        g = greenList.getSelectedIndex ();

        if (r != -1 && g != -1 && b != -1)
            getColorSelectionModel ().setSelectedColor (
                new Color (r, g, b));
    }

    public String getDisplayName () {

```

```

    return "display name";
}

public Icon getSmallDisplayIcon () {
    return null;
}

public Icon getLargeDisplayIcon () {
    return null;
}
}

```

组件总结 16-2 总结了 JColorChooser 类。

组件总结 16-2 JColorChooser

模型: javax.swing.colorchooser.ColorSelectionModel
 UI 代表: javax.swing.plaf.basic.BasicColorChooserUI
 绘制器: _____
 编辑器: _____
 激发的事件: ActionEvents/ChangeEvents/PropertyChangeEvents
 替换: _____
 类图:

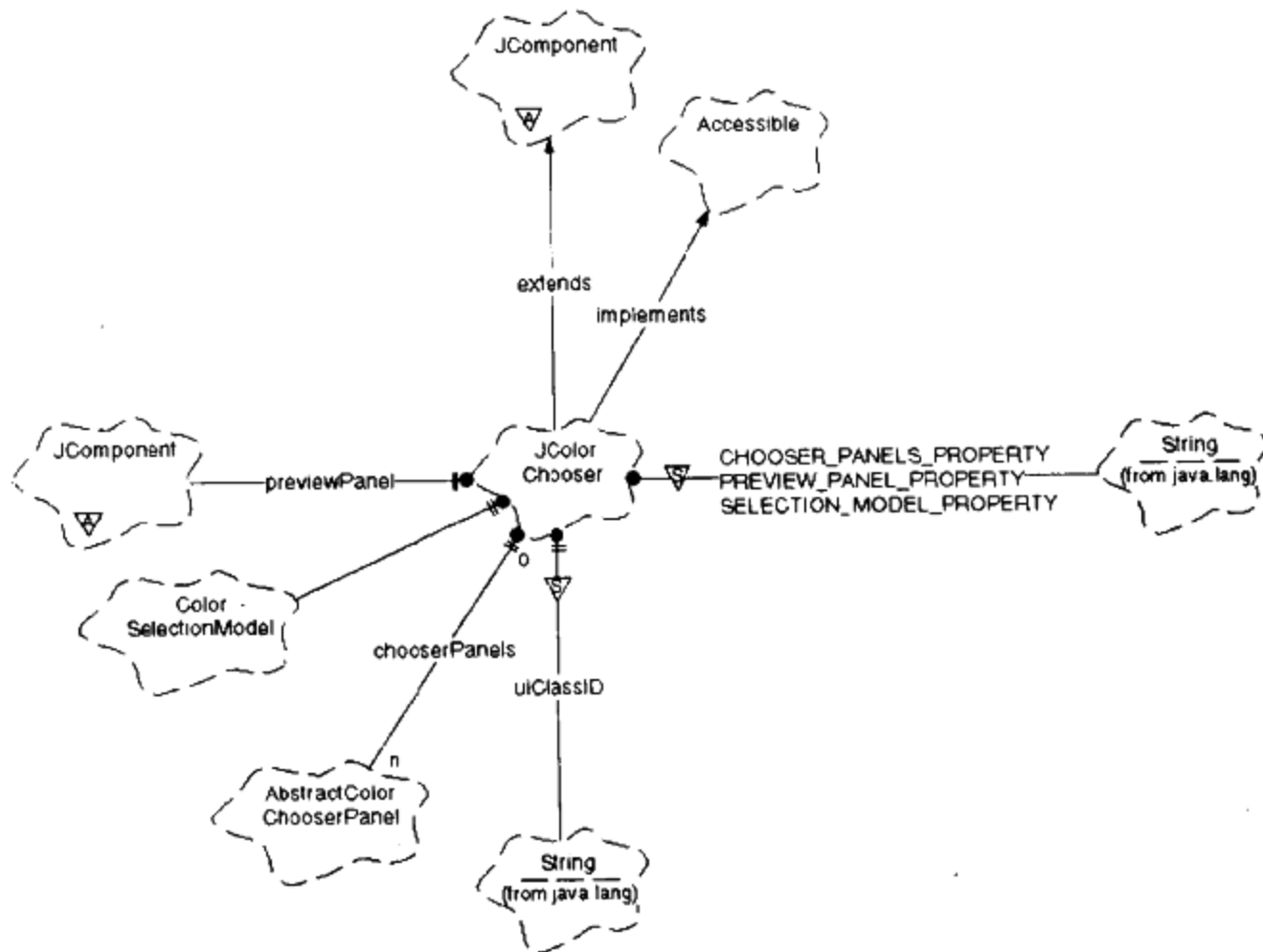


图 16-15 JColorChooser 类图

JColorChooser 扩展 JComponent 并实现 Accessible 接口。JColorChooser 维护对它的预览面板 (JComponent 的一个实例) 的 protected 引用, 还维护了对一个颜色选取面板数组的 protected 引用, 这些颜色选取面板是实现 AbstractColorChooserPanel 类的对象。另外, JColorChooser 类定义一组 public static 字符串, 用它们来表示与 JColorChooser 的实例有关的属性。

16.2.3 JColorChooser 属性

表 16-4 列出了由 JColorChooser 类维护的属性。

表 16-4 JColorChooser 属性

属性名	数据类型	属性类型 ^①	访问 ^③	缺省 ^④
chooserPanels	AbstractColorChooserPanel []	B	SG	L&F
color	Color	S ^②	CSG	L&F
previewPanel	JComponent	B	SG	L&F
selectionModel	ColorSelectionModel	B	CSG	L&F

- ① B = 关联的 (激发 PropertyChangeEvent) / C = 受约束的 / I = 索引的 / S = 简单的 / Ch = 激发 ChangeEvent / LD = 激发 ListDataEvent
- ② 当颜色被改变时, 选取模型激发一个变化事件
- ③ C = 可在创建时设置 / G = 获取方法 / S = 设置方法
- ④ L&F = 与界面样式有关

chooserPanels——在颜色选取器中显示的那些面板, 它们用于选取一种颜色。
color——颜色选取器中当前选取的颜色。
previewPanel——实时显示当前所选取的颜色的面板。可以用 JColorChooser.setPreviewPanel 方法来替换缺省的预览面板。有关替换缺省预览面板的更多信息, 请参见 16.2.2 节。
selectionModel——实现 swing.colorchooser.ColorSelectionModel 接口的选取模型。在当前所选取的颜色被修改时, 颜色选取器选取模式将激发变化事件。

16.2.4 JColorChooser 事件

当清除由 JColorChooser.createDialog () 创建的非模态颜色选取器对话框时, 处理颜色选取器事件最常用的办法是激发动作事件。不能为包含在模态颜色选取器对话框中的按钮指定监听器, 模态颜色选取器对话框是由 JColorChooser.showDialog () 创建的。

表 16-5 列出了由组成颜色选取器的类激发的有意义的事件。

表 16-5 颜色选取器事件

事件	由 ... 激发	触发/处理
Action	JColorChooser	<u>触发</u> : OK/Cancel 按钮
Change	DefaultColor SelectionModel	<u>触发</u> : 颜色选取, <u>处理</u> : UI 代表设置预览面板的前景色
Property Change	JColorChooser	<u>处理</u> : UI 代表对预览面板作出反应而且选取器面板变化了

当修改选取的颜色时, 颜色选取器的缺省颜色选取模型激发变化事件。文件选取器的 UI 代表对来自选取模型的变化事件作出反应, 即通过把预览面板的前景色设置为所选取的颜色并重新绘制这个面板。当修改这些颜色选取器的关联属性时, 这些颜色选取器将激发属性变化事件。要对颜色选取器的 color 属性的变化作出反应, 则必须向选取器的选取模型登记一个变化监听器, 如例 16-9 所示。

16.2.5 JColorChooser 类总结

类总结 16-6 列出了 JColorChooser 的 public 和 protected 变量和方法。

类总结 16-6 JColorChooser

1. 常量

```
public static final String CHOOSE_PANELS_PROPERTY
```

```
public static final String PREVIEW_PANEL_PROPERTY
```

```
public static final String SELECTION_MODEL_PROPERTY
```

由 JColorChooser 类定义的这些常量识别颜色选取器属性。

2. 构造方法

```
public JColorChooser ()
```

```
public JColorChooser (Color)
```

```
public JColorChooser (ColorSelectionModel)
```

JColorChooser 类提供上面所列的三个构造方法。无参数构造方法创建一个初始颜色为白色的颜色选取器。还可以用上面所列的最后两种构造方法来显式地指定初始颜色，这两种构造方法分别以一个颜色或颜色选取模型为参数。

3. 方法

(1) 创建和显示颜色选取器对话框

```
public static JDialog createDialog (Component, String, boolean, JColorChooser,  
                                   ActionListener, ActionListener)
```

```
public static Color showDialog (Component, String, Color)
```

上面所列的这些 static 方法用于创建或显示一个包含 JColorChooser 的一个实例的对话框。“在对话框中显示颜色选取器”对这些方法进行了介绍。

(2) 选取器面板

```
public void addChoosePanel (AbstractColorChooserPanel)
```

```
public AbstractColorChooserPanel removeChooserPanel (AbstractColorChoosePanel)
```

```
public AbstractColorChooserPanel [] getChooserPanels ()
```

```
public void setChooserPanels (AbstractColorChooserPanel [])
```

颜色选取器允许完全控制在它们上面显示的颜色选取器面板。可以使用上面所列的前两个方法把面板添加到颜色选取器面板数组中或从颜色选取器面板数组中删除面板，这个颜色选取器面板数组由 JColorChooser 的实例来维护。

可以分别用上面所列的最后两个方法来访问整个颜色选取器面板数组和设置这个数组。要了解替换存储在一个颜色选取器中的颜色选取器面板的一个样例，请参见 16.2.3 节。

(3) 颜色和颜色选取模型

```
public Color getColor ()
```

```
public void setColor (int colorBits)
```

```
public void setColor (int red, int green, int blue)
```

```
public void setColor (Color)
```

```
public ColorSelectionModel getSelectionModel ()
```

```
public void setSelectionModel (ColorSelectionModel)
```

可以用上面所列的这些 setColor 方法来显式地控制颜色选取器中所选取的颜色。可以用一个 integer 值来指定一个颜色，这个 integer 值的低字节指定蓝色组件、中间的字节指定绿色组件，高字节指定红色组件。

设置颜色选取器的颜色会引起它的选取模型激发一个变化事件。例 16-9 是颜色选取器的颜色变化的例子。

JColorChooser 有一个实现 swing.colorchooser.ColorSelectionModel 接口的颜色选取模型。可以

在构造后用上面所列最后两个方法来获取或设置一个颜色选取器的选取模型。

(4) 预览面板

```
public JComponent getPreviewPanel ()
```

```
public void setPreviewPanel (JComponent)
```

颜色选取器显示一个预览面板，这个预览面板实时显示当前选取的颜色。可以用上面所列的最后一个方法来替换标准的预览面板。

(5) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
```

```
public ColorChooserUI getUI ()
```

```
public String getUIClassID ()
```

```
public void setUI (ColorChooserUI)
```

```
public void updateUI ()
```

上面列出的方法可以在大多数 *JComponent* 扩展中找到。*Swing* 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。*updateUI* 方法在组件配备了 UI 代表时调用。

16.2.6 AWT 兼容

AWT 没有提供与 *JColorChooser* 类似的组件。

16.3 本章回顾

Swing 的文件选取器和颜色选取器都是现代图形用户界面中很重要的、可配置的组件。可以用许多方法来定制文件选取器，包括使用可访问组件和文件过滤器。这两个选取器都是显示在对话框中的。

对 *Swing* 1.1 FCS 而言，目前还没有实现文件选取器的多文件选取，而且，由于有错误，还不能为颜色选取器设置预览面板。

第 17 章 列 表

由 JList 类代表的 Swing 列表显示一个可选取对象列表，它支持三种选取模式：单选取、单间隔选取和多间隔选取。

JList 类把维护和绘制列表的工作委托给一个对象来完成。一个列表的模型维护一个对象列表，列表单元绘制器将这些对象绘制在列表单元中。

缺省情况下，列表单元绘制器是 DefaultListCellRenderer 的实例，它绘制表 17-1 中列出的对象。图标和字符串按原样显示，而所有其他类型对象的绘制方式是：通过显示从这些对象的 toString 返回的字符串来绘制这些对象。

表 17-1 DefaultListCellRenderer 绘制

对象类型...	...被绘制
图标	按原样显示
对象	用从 toString () 返回的字符串
字符串	按原样显示

与按钮和标签不同，列表单元在缺省情况下可以显示一个字符串或一个图标，但是不能将它们同时显示，因为 DefaultListCellRenderer 只显示一个表 17-1 中列出的对象。幸运的是，实现一个定制的绘制器（它可以绘制所需的任何类型的对象）是一个简单的事情。

JList 本身没有对它显示的对象提供进行滚动的支持，但是可以把 JList 的实例放在一个滚动窗格中。有关 Swing 滚动窗格的更多信息，请参见 13.2 节“JScrollPane”。

JList 组件把三个主要的工作交给其他的对象来完成，这三个工作是：数据处理、项选取和单元绘制，它们分别由实现 ListModel、ListSelectionModel 和 ListCellRenderer 接口的那些类来处理。

图 17-1 所示的小应用程序包含一个 JList 实例，这个实例包含 10 个字符串。

例 17-1 列出了图 17-1 所示的小应用程序的代码。

例 17-1 一个简单的列表样例

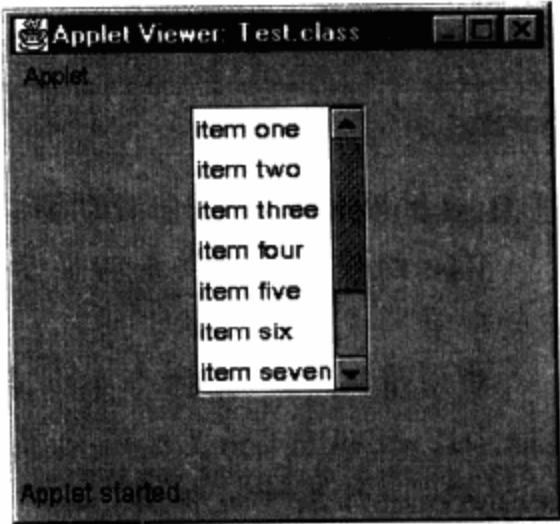


图 17-1 一个简单的列表样例

```
import java.awt.* ;
import javax.swing.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane () ;
        Object [] items = { "item one", "item two", "item three",
                           "item four", "item five", "item six",
                           "item seven", "item eight",
                           "item nine", "item ten" } ;
```



```

JList list = new JList (items);
JScrollPane sp = new JScrollPane (list);
list.setVisibleRowCount (7);
contentPane.setLayout (new FlowLayout ());
contentPane.add (sp);

```

这个列表是用一个字符串数组为参数来创建的，并且放在一个滚动窗格中。通过调用 `JList.setVisibleRowCount` 方法把滚动窗格中显示的可见行数设置为 7，然后把这个滚动窗格添加到这个小应用程序的内容窗格中。在列表中显示的字符串被指定为一个对象数组，以强调这样一个事实：`JList` 的实例可以显示任何类型的对象。列表在缺省时只能显示图标或字符串，所有其他类型的对象都用从这些对象的 `toString` 方法返回的字符串来代表。

缺省情况下，当一个列表放在一个滚动窗格中时，列表中有 8 行在滚动窗格中是可见的。当列表包含在一个滚动窗格中时，可以用 `JList.setVisibleRowCount` 方法来设置可见行数。只有当列表在滚动窗格中显示时，`setVisibleRowCount` 方法才起作用。

Swing 提示

把 `JList` 的实例放在一个滚动窗格中

`JList` 组件没有提供滚动显示对象的方法。另外，经常要调整 `JList` 实例的大小，以便列表中的所有项是不可见的。所以，几乎总是把列表放在一个滚动窗格中。

缺省情况下，当把一个列表放在一个滚动窗格中时，列表中有 8 行是可见的。可以用 `JList.setVisibleRowCount` 方法来显式地设置可见行数。只有当滚动窗格包含列表时，`setVisibleRowCount` 方法才起作用。

17.1 列表模型

`JList` 类不维护对它所显示的那些对象的引用。`JList` 的所有实例都把它们的数据管理工作委托给一个实现 `ListModel` 接口的对象。

当用下面的 `JList` 构造方法来构造 `JList` 的一个实例时，可以指定列表中所显示的对象：

- `public JList (ListModel)`
- `public JList (Object [])`
- `public JList (Vector)`

还可以在使用下面的 `JList` 方法构造列表后，再指定列表中要显示的对象：

- `public void setModel (ListModel)`
- `public void setListData (Object [])`
- `public void setListData (Vector)`

除上面所列的这些构造方法和方法外，`JList` 类本身没有对它的数据提供操纵方法。例如，没有插入或删除对象的 `JList` 方法。所以，必须通过直接操纵列表模型来插入或删除列表数据（除替换列表中的所有数据外）。

所有的列表模型都实现 `ListModel` 接口，接口总结 17-1 总结了 `ListModel` 接口。

接口总结 17-1 `ListModel`

1. ListDataListener 登记

```
public abstract void addListDataListener (ListDataListener)
public abstract void removeListDataListener (ListDataListener)
```

ListModel 接口定义添加和删除 ListDataListener 实例的方法，当修改由列表模型维护的数据时，将通知这些实例。有关 ListDataListener 接口及它的使用的更多信息，请参见接口总结 17-5。

2. 列表大小和元素的访问方法

```
public abstract int getSize ()
public abstract Object getElementAt (int)
```

ListModel 接口还定义获取列表中所显示的对象数量的方法——getSize () 和获取对列表中一个指定对象的引用的方法——getElementAt ()。

Swing 提供了两个实现 ListModel 接口的类：AbstractListModel 和 DefaultListModel，如图 17-2 所示。

AbstractListModel 扩展 Object 并实现 ListModel 接口和 Serializable 接口。AbstractListModel 实现在模型接口中定义的方法，通过维护一个 EventListenerList 来登记 ListDataListeners。AbstractListModel 不实现那些由 ListModel 接口定义的、与模型的数据有关的方法。

DefaultListModel 扩展 AbstractListModel 并实现 ListModel 方法，这些方法是超类没有实现的方法。DefaultListModel 以矢量的形式存储数据。

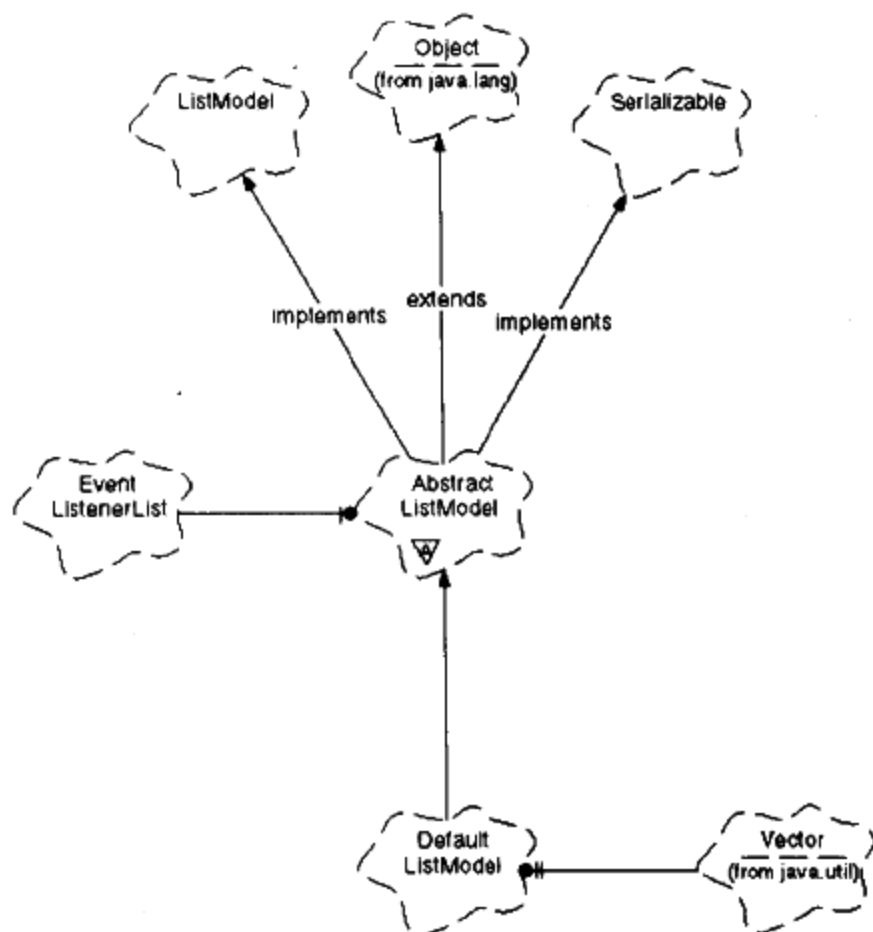


图 17-2 列表模型

17.1.1 AbstractListModel

AbstractListModel 不关心存储数据的问题，因此没有实现由 ListModel 接口定义的 getSize 和 getElementAt 方法。由于 AbstractListModel 没有实现由 ListModel 接口定义的所有方法，所以它是一个抽象类。

AbstractListModel 类扩展 Object 并实现 ListModel 接口和 Serializable 接口。DefaultListModel 类是 AbstractListModel 的一个扩展。

类总结 17-1 总结了 AbstractListModel 类。

类总结 17-1 AbstractListModel

扩展：java.lang.Object

实现：ListModel、java.io.Serializable

1. 构造方法

```
public AbstractListModel ()
```

这个无参数的构造方法是由编译器提供的，AbstractListModel 没有实现任何的构造方法。

2. 方法

```

public void addListDataListener (ListDataListener)
protected void fireContentsChanged (Object source, int index0, int index1)
protected void fireIntervalAdded (Object source, int index0, int index1)
protected void fireIntervalRemoved (Object source, int index0, int index1)
public void removeListDataListener (ListDataListener)

```

AbstractListModel 只把列表数据事件发送给监听器；它实现在 ListModel 接口中定义的 addListDataListener 和 removeListDataListener 方法。AbstractListModel 还提供三个 protected 方法,当修改列表内容、向列表数据添加间隔或从列表数据删除间隔时,这三个方法把事件发送给监听器。

当列表中显示的对象被指定为一个对象数组或一个矢量时,就把 AbstractListModel 的一个内部扩展实例化并把它与这个列表相关联。例如, JList.setListData (Object []) 和 JList.setListData (Vector) 的实现如下所示:

```

//From the JList class...

public void setListData (final Object [] listData) {
    setModel (new AbstractListModel () {
        public int getSize () {
            return listData.length;
        }
        public Object getElementAt (int i) {
            return listData [i];
        }
    });
}

public void setListData (final Vector listData) {
    setModel (new AbstractListModel () {
        public int getSize () {
            return listData.size ();
        }
        public Object getElementAt (int i) {
            return listData.elementAt (i);
        }
    });
}

```

当指定列表数据为一个对象数组或一个矢量时,就知道了存储数据的方式,因此,可以实现 ListModel 接口的 getSize 和 getElementAt 方法。

把列表数据指定为一个对象数组或一个矢量没有提供插入和删除对象的能力。如前所述, JList 类没有提供插入或删除对象的方法,这可以从类总结 17-1 中看到, AbstractListModel 类也不提供插入或删除对象的方法。如果必须把对象插入到列表模型中或从列表模型中删除,则 JList 必须显式地使用 DefaultListModel 的一个实例或它的一个扩展。

17.1.2 DefaultListModel

DefaultListModel 类是 AbstractListModel 的一个具体 (非抽象) 扩展, DefaultListModel 实现 java.util.Vector API, 并把数据存储和获取的工作委托给 Vector 的一个实例。类总结 17-2 总结了 DefaultListModel 类。

类总结 17-2 DefaultListModel

扩展: AbstractListModel

1. 构造方法

```
public DefaultListModel ()
```

DefaultListModel 的这个无参数构造方法是由编译器产生的。

2. 方法

```
public void add (int, Object)
public void addElement (Object)
public int capacity ()
public void clear ()
public boolean contains (Object)
public void copyInto (Object [])
public Object elementAt (int)
public Enumeration elements ()
public void ensureCapacity (int minCapacity)
public Object firstElement ()
public Object get (int)
public Object getElementAt (int index)
public int getSize ()
public int indexOf (Object)
public int indexOf (Object element, int)
public void insertElementAt (Object element, int index)
public boolean isEmpty ()
public Object lastElement ()
public int lastIndexOf (Object element)
public int lastIndexOf (Object element, int index)
public Object remove (int index)
public void removeAllElement ()
public boolean removeElement (Object element)
public void removeAllElementAt (int index)
public void removeRange (int index0, int index1)
public Object set (int index, Object element)
public void setElementAt (Object element, int index)
public void setSize (int size)
public int size ()
public Object [] toArray ()
public String toString ()
public void trimToSize ()
```

DefaultListModel 实现 ListModel 接口中的 getSize 和 getElementAt 方法，因此，它不是抽象类。由 DefaultListModel 实现的其余方法复制 java.util.Vector API，并把操作委托给一个 Vector。注意，在图 17-2 中，这个矢量是 DefaultListModel 类的一个 private 成员，因为在将来的 Swing 版本中这个矢量可能会变成一个集合。

DefaultListModel 类最重要的方面是提供了插入和删除的方法。如果必须向一个列表插入一些对象或从列表中删除一些对象，则这个列表应该配备一个 DefaultListModel 实例或配备 AbstractListModel 的一个提供插入和删除方法的定制扩展。

图 17-3 所示的小应用程序包含一个配备了 DefaultListModel 实例的列表。这个小应用程序提供删除当前所选项的按钮和向列表添加一个项的按钮。

图 17-3 左上角的图片显示了列表中有两个选取项的小应用程序。右上角的图片显示了激活 remove selected items 按钮后小应用程序的样子，左下角的图片显示激活 add item 按钮并显示

输入对话框后小应用程序的样子，右下角的图片显示在向列表添加一项后小应用程序的样子。

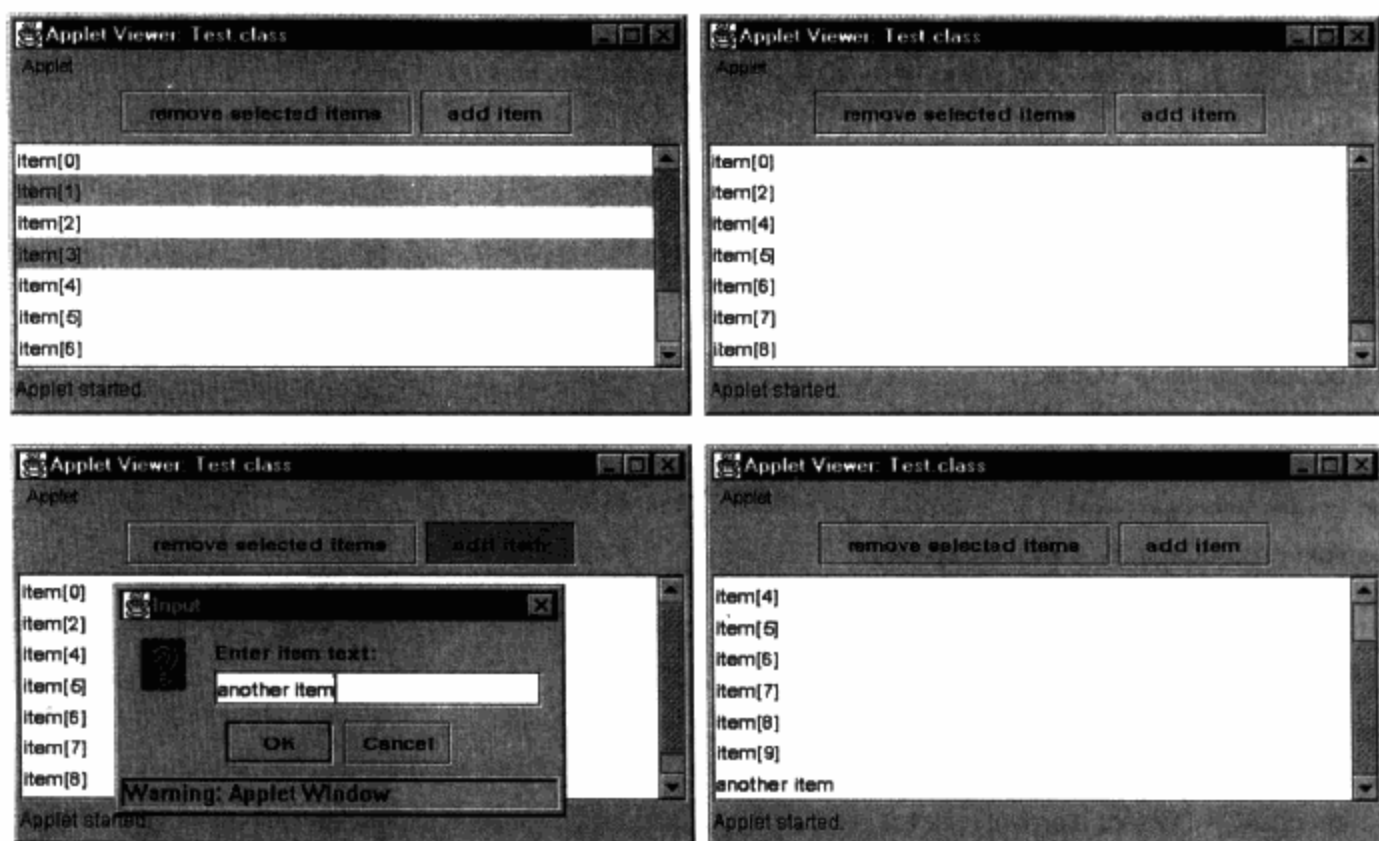


图 17-3 一个具有缺省列表模型的列表

这个小应用程序用 `JList` 的无参数构造方法来创建一个列表，将这个列表放在一个滚动窗格中。把 `ControlPanel` 的一个实例（一个 `JPanel` 扩展，它包含这个小应用程序的按钮）实例化，并且把控制面板和滚动窗格都添加到这个小应用程序的内容窗格中。

将 `DefaultListModel` 的一个实例实例化，而且把数组中的每个字符串都添加到这个模型中。然后，用 `JList.setModel` 方法把这个模型指定为这个列表的模型。

```
public class Test extends JApplet {
    private JList list = new JList ();

    String [] items = { "item [0]", "item [1]", "item [2]",
                        "item [3]", "item [4]", "item [5]",
                        "item [6]", "item [7]",
                        "item [8]", "item [9]" };

    public void init () {
        // create controlpanel and add list and control panel
        // to applet's content pane ...

        populateList ();
    }

    public void populateList () {
        DefaultListModel model = new DefaultListModel ();

        for (int i=0; i < items.length; ++i)
            model.addElement (items [i]);

        list.setModel (model);
    }
}
```

当把删除选取项的按钮激活时，通过调用 `JList.getSelectedIndices` 方法来获得列表中的选取索引，并且通过调用 `JList.getModel()` 来获得对列表模型的一个引用。对列表中的每个选取索引都调用 `DefaultListModel.removeElementAt`，以便从列表中删除相应的字符串。

```

class ControlPanel extends JPanel {
    JButton remove = new JButton ("remove selected items");
    JButton add = new JButton ("add item");

    public ControlPanel (final JList list) {
        add (remove);
        add (add);
        remove.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int [] selected = list.getSelectedIndices ();
                DefaultListModel model =
                    (DefaultListModel) list.getModel ();
                for (int i=0; i < selected.length; ++i) {
                    model.removeElementAt (selected [i] - i);
                }
            }
        });
    }
    ...
}

```

当激活了添加项的按钮时，调用 `JOptionPane.showInputDialog ()` 来获得添加到列表中的字符串。有关 `JOptionPane` 和输入对话框的更多信息，请参见 14.3 节 “`JOptionPane`”。

```

...
add.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        final DefaultListModel model =
            (DefaultListModel) list.getModel ();

        String s = JOptionPane.showInputDialog (
            list, "Enter item text:");

        model.addElement (s);

        SwingUtilities.invokeLater (new Runnable () {
            public void run () {
                list.ensureIndexIsVisible (
                    model.getSize () -1);
            }
        });
    }
});
}

```

清除这个对话框后，把对话框文本域中的字符串添加到列表的模型中。在添加字符串到模型中后，`JList.ensureIndexIsVisible` 被调用以确保新添加的项是可见的。对 `ensureIndexIsVisible` 的调用封装在一个 `Runnable` 对象中，并通过 `SwingUtilities.invokeLater ()` 来执行，有关 `SwingUtilities.invokeLater ()`，请参见 2.4 节 “Swing 与线程”。使用 `invokeLater ()` 方法的原因如下：

把数据添加到一个列表模型中将调用列表的 `revalidate ()`。`revalidate` 方法把一个事件放在事件队列上，这引起列表重新调整大小以便容纳这个新数据，有关 `revalidate` 方法的更多信息，请参见 4.3.5 节 “`validate`、`invalidate` 和 `revalidate` 方法”。

所以，图 17-3 所示的列表将不能重新调整大小，直到对 `addActionListener` 的调用返回。如果在 `addActionListener` 方法中直接调用 `JList.ensureIndexIsVisible ()`，则列表的倒数第二项将是可见的，因为这个列表还没有重新调整大小以容纳添加到这个列表中的项。把对 `ensureIndexIsVisible ()` 的调用封装在 `Runnable` 中确保了在把列表重新调整大小事件处理后对 `ensureIndex-`

isVisible 的调用，Runnable 被传送给 SwingUtilities.invokeLater()。

例 17-2 列出了图 17-3 所示的小应用程序的代码。

例 17-2 一个带有缺省列表模型的列表

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {
    private JList list = new JList();

    String [] items = { "item [0]", "item [1]", "item [2]",
                        "item [3]", "item [4]", "item [5]",
                        "item [6]", "item [7]",
                        "item [8]", "item [9]" };

    public void init () {
        Container contentPane = getContentPane();
        JPanel controlPanel = new ControlPanel (list);

        contentPane.add (controlPanel, BorderLayout.NORTH);
        contentPane.add (new JScrollPane (list),
                        BorderLayout.CENTER);

        populateList ();
    }

    public void populateList () {
        DefaultListModel model = new DefaultListModel ();

        for (int i=0; i < items.length; ++i)
            model.addElement (items [i]);
        list.setModel (model);
    }
}

class ControlPanel extends JPanel {
    JButton remove = new JButton ("remove selected items");
    JButton add = new JButton ("add item");

    public ControlPanel (final JList list) {
        add (remove);
        add (add);

        remove.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int [] selected = list.getSelectedIndices ();
                DefaultListModel model =
                    (DefaultListModel) list.getModel ();

                for (int i=0; i < selected.length; ++i) {
                    model.removeElementAt (selected [i] - i);
                }
            }
        });

        add.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                final DefaultListModel model =
                    (DefaultListModel) list.getModel ();

                String s = JOptionPane.showInputDialog (
                    list,
                    "Enter item text:");
            }
        });
    }
}
```



```
model.addElement (s);  
SwingUtilities.invokeLater (new Runnable () {  
    public void run () {  
        list.ensureIndexIsVisible (  
            model.getSize () -1);  
    }  
});  
}  
};
```

Swing 提示

指定一个明确的列表模型

通过在构造时或之后的任何时刻指定一个对象数组、一个矢量或一个列表模型，可以把数据与 JList 的实例相关联。指定一个对象数组或一个矢量将创建一个简单的 AbstractListModel 扩展，然后把这个扩展与所需的列表相关联。

指定一个列表模型需要将一个 DefaultListModel 实例（或一个 AbstractListModel 或 DefaultListModel 的定制扩展）进行实例化，然后把这个模型传送给 JList.setModel () 或相应的 JList 构造方法。通常，由于下述原因之一，需要显式地指定列表模型：

- 1) 必须向列表中插入项或从列表中删除项。
- 2) 除用一个矢量或一个对象数组表示数据外，数据必须用数据结构来表示。

17.2 列表选取

除把数据管理工作交给列表模型外，JList 类还把跟踪所选项的工作交给列表选取模型。JList 类把选取管理的工作交给实现 ListSelectionModel 接口的对象。列表选取模型支持三种选取模式，如图 17-4 所示，从左至右分别是：单选取、单间隔选取和多间隔选取。

单选取模式在给定的任意时刻都只允许选取列表中的一项。单间隔选取模式允许选取多个项，但是所选取的项必须是相邻的。多间隔选取模式允许选取多个单间隔选取。JList 实例的缺省选取模式是多间隔选取。接口总结 17-2 总结了 ListSelectionModel 接口。

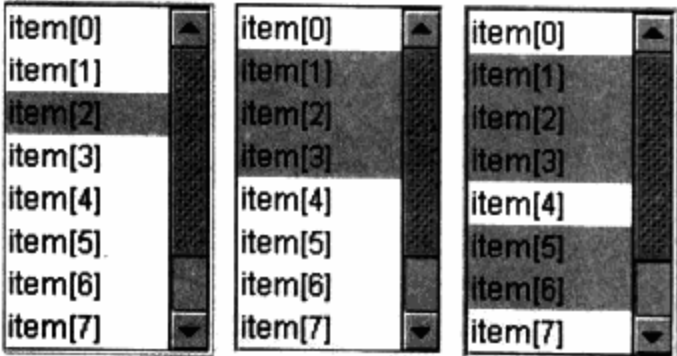


图 17-4 ListSelectionModel 选取模式

接口总结 17-2 ListSelectionModel

1. 常量

```
public static final int MULTIPLE_INTERVAL_SELECTION  
public static final int SINGLE_INTERVAL_SELECTION  
public static final int SINGLE_SELECTION
```

ListSelectionModel 接口为它的选取模式定义常量。ListSelectionModel 还定义了添加和删除选取间隔、清除选取的方法，还定义了设置和获得 anchor、lead、max、min、valueAdjusting 等选取属性的方法。JList 选取属性将在 17.3.1 节“JList 属性”中介绍。

2. 方法[⊖]

```
public abstract void addSelectionInterval (int, int)
public abstract void clearSelection ()
public abstract int getAnchorSelectionIndex ()
public abstract int getLeadSelectionIndex ()
public abstract int getMaxSelectionIndex ()
public abstract int getMinSelectionIndex ()
public abstract int getSelectionMode ()
public abstract boolean getValuesAdjusting ()
public abstract void insertIndexInterval (int, int, boolean)
public abstract boolean isSelectionIndex (int)
public abstract boolean isSelectionEmpty ()
public abstract void removeIndexInterval (int, int)
public abstract void removeListSelectionListener (ListSelectionListener)
public abstract void removeSelectionInterval (int index0, int index1)
public abstract void setAnchorSelectionIndex (int)
public abstract void setLeadSelectionIndex (int)
public abstract void setSelectionInterval (int index0, int index1)
public abstract void setSelectionMode (int)
public abstract void setValuesAdjusting (boolean)
```

缺省情况下，JList 的实例配备一个 DefaultListSelectionModel 实例，它是由 Swing 提供的 ListSelectionModel 接口的唯一实现。图 17-5 示出了 DefaultListSelectionModel 类的类图。

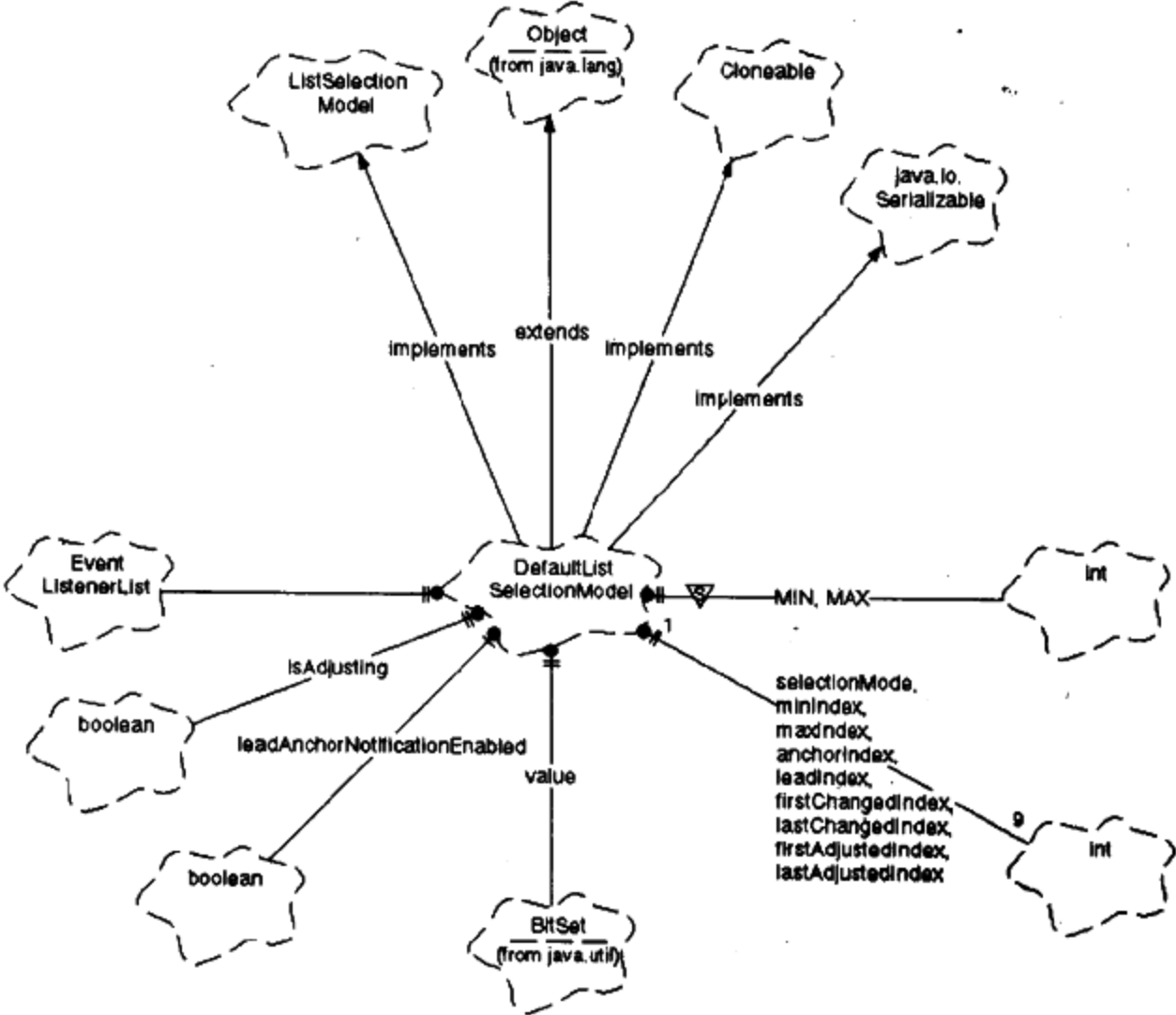


图 17-5 DefaultListSelectionModel 类

DefaultListSelectionModel 类扩展 Object 并实现 Cloneable 接口、Serializable 接口和 ListSelection-

⊖ 用斜体字表示的方法由 Jlist 类实现

Model 接口。DefaultListSelectionModel 维护许多跟踪选取状态的 private 引用，还维护一个跟踪列表选取监听器的 EventListenerList。

与列表模型不同，很少由开发人员来实现或直接维护列表选取模型，原因有两个。

首先，缺省列表选取模型提供的三种选取模式几乎满足了 JList 类的所有应用。因此，开发人员不需要实现他们自己的列表选取模型以便提供另外一些选取模式。这是与定制列表模型不同的，定制列表模型经常实现以便允许向列表中插入项或从列表中删除项。

其次，几乎所有由 ListSelectionModel 接口定义的方法都由 JList 类交给列表选取模型来实现。因此，与列表模型不同，不需要直接访问列表选取模型来修改列表的选取状态。由 ListModel 接口定义的方法没有由 JList 类实现。

要了解如何处理列表选取事件，请参见 17.3.2 节。

Swing 提示

列表模型与列表选取模型的比较

JList 类有两个模型：列表模型和列表选取模型，它们分别由 ListModel 接口和 ListSelectionModel 接口定义。可以用 JList.getModel 和 JList.getSelectionModel 方法来访问这两个模型。

通过委托给 JList 类的选取模型，JList 类实现了所有由 ListSelectionModel 接口定义的方法。因此，可以操纵选取模型，而不用由 JList 的方法直接访问。相反，JList 类不实现由 ListModel 接口定义的方法，因此，只有通过访问列表模型才可以直接操纵列表数据。

列表模型与列表选取模型的另一个差别是缺省的列表选取模型 (ListSelectionModel) 比缺省的列表模型 (AbstractListModel) 使用的频率高。因为 AbstractListModel 没有提供插入项和删除项的方法，所以经常要显式地指定列表模型。相反，DefaultListSelectionModel 几乎提供了大多数列表所需的所有操作，因此，很少实现定制的列表选取模型。

17.3 列表单元绘制器

除委托数据管理和选取工作外，JList 类还把其单元的绘制工作交给另一个对象去做。JList 的每个实例都维护对这样的一个对象的引用，这个对象实现 ListCellRenderer 接口，接口总结 17-3 总结了 ListCellRenderer。

接口总结 17-3 ListCellRenderer

ListCellRenderer 接口只定义了一个方法，这个方法返回一个组件：

```
public abstract Component getListCellRendererComponent (JList list,
                                                         Object value,
                                                         int index,
                                                         boolean isSelected,
                                                         boolean cellHasFocus)
```

由 getListCellRendererComponent 返回的组件的作用就像一个橡皮图章，它把这个组件绘制到列表中项所占的区域。注意列表单元不包含这个组件，这个组件只是绘制到列表单元上。这种差别是很重要的，因为不能操纵这个组件，只能使用这个组件的可见代表来绘制列表单元。

getListCellRendererComponent 方法带入一个列表、与一个单元相关联的值、这个单元的索引，这个单元是否被选取了的信息，和这个单元是否有焦点的信息。

缺省情况下，JList 的实例配备一个绘制器，它是 ListCellRenderer 接口的一个简单实现，即

DefaultListCellRenderer 类。DefaultListCellRenderer 类扩展 JLabel 类，而且可以显示字符串或图标，但不能在一个单元中同时显示字符串和图标。除字符串或图标外的对象显示由这个对象的 toString 方法返回的字符串。如果需要以不同的形式绘制列表单元，则必须实现一个定制列表单元绘制器，并且这个绘制器的实例还必须与所讨论的列表相关联。

类总结 17-3 总结了 DefaultListCellRenderer 类。

类总结 17-3 DefaultListCellRenderer

扩展：JLabel

实现：ListCellRenderer、java.io.Serializable

1. 构造方法

```
public DefaultListCellRenderer ()
```

DefaultListCellRenderer 只提供一个无参数的构造方法并实现 getListCellRendererComponent 方法以便返回绘制器本身，如前所述，DefaultListCellRenderer 扩展 JLabel。

2. 方法

```
public Component getListCellRendererComponent (JList list,
                                                Object value,
                                                int index,
                                                boolean isSelected,
                                                boolean cellHasFocus)
```

DefaultListCellRenderer 维护一个 static noFocusBorder，它是一个空边框 (swing.border.EmptyBorder 的一个实例)，它在绘制器边缘占据一个像素的空间。这个边框是 static 的，因为一个边框可以由许多组件所共享，因此，DefaultListCellRenderer 的所有实例都使用同一个边框。有关共享边框的更多信息，请参见 5.1.7 节“边框库——共享边框”。

也许不需要声明，但当绘制器没有焦点时，就使用 noFocusBorder。当绘制器没有焦点时，则从 UIManager 类中获得一个边框：

```
// from DefaultListCellRenderer.getDefaultListCellRendererComponent ()
setBorder ( (cellHasFocus) ?
    UIManager.getBorder ("List.focusCellHighlightBorder") : noFocus-
    Border);
```

有关 UIManager 类、缺省边框和颜色的安装的更多信息，请参见 7.1.3 节“UI 管理器”。

图 17-6 所示的小应用程序包含一个列表，这个列表配备了可以在同一个单元同时显示一个图标和一个字符串的定制列表单元绘制器。这个小应用程序实现了定制列表模型，它与定制绘制器同步工作。

这个小应用程序创建两个字符串数组，一个数组代表在单元中显示的字符串，另一个数组代表用作图标的图像文件名。

这个小应用程序还创建了一个 NameAndPictureListModel 实例和一个 NameAndPictureListCellRenderer 实例，并分别把这两个实例指定为列表模型和绘制器。

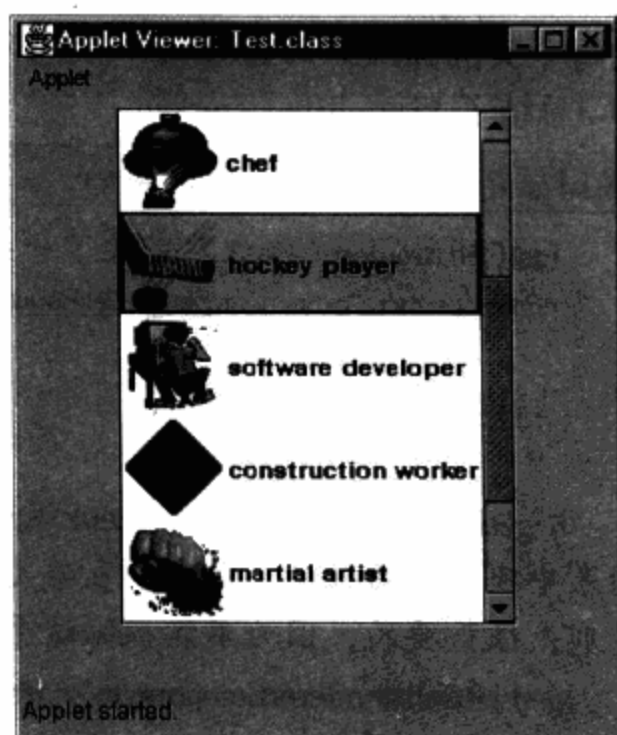


图 17-6 实现定制列表单元绘制器

```

public class Test extends JApplet {
    private String [] names = new String [] {
        ...
    };
    private String [] pics = new String [] {
        ...
    };

    public void init () {
        Container contentPane = getContentPane ();

        ListModel model =
            new NameAndPictureListModel (names, pics);

        ListCellRenderer renderer =
            new NameAndPictureListCellRenderer ();

        JList list = new JList (model);
        list.setCellRenderer (renderer);
        list.setVisibleRowCount (5);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (new JScrollPane (list));
    }
}

```

NameAndPictureListModel 类扩展 DefaultListModel，并以一个名字数组和一个图像文件名数组为参数来构造。用每个文件名来构造图标，名字或图标对作为带两个元素的 Object 数组存储在这个模型中。

NameAndPictureListModel 类还提供了访问这些名字和图标的访问方法，这些访问方法以给定的一个 Object（代表存储在模型中的一项）为参数。有了这些访问方法，绘制器就不需要知道名字和图标是如何存储在这个模型中的。

```

class NameAndPictureListModel extends DefaultListModel {
    public NameAndPictureListModel (String [] names,
                                    String [] pics) {
        for (int i=0; i < names.length; ++i) {
            addElement (new Object [] {
                names [i], new ImageIcon (pics [i]) });
        }
    }

    public String getName (Object object) {
        Object [] array = (Object []) object;
        return (String) array [0];
    }

    public Icon getIcon (Object object) {
        Object [] array = (Object []) object;
        return (Icon) array [1];
    }
}

```

NameAndPictureListCellRenderer 类扩展 JLabel，并实现 ListCellRenderer 接口。它创建了两个边框，边框的使用取决于一个列表单元是否有焦点。

NameAndPictureListCellRenderer 构造方法把不透明属性设置为 true，因为缺省时，JLabel 的实例是部分透明的。

```

class NameAndPictureListCellRenderer extends JLabel
    implements ListCellRenderer {

    private Border

```

```

lineBorder = BorderFactory.createLineBorder (Color.red, 2),
emptyBorder = BorderFactory.createEmptyBorder (2, 2, 2, 2);

public NameAndPictureListCellRenderer () {
    setOpaque (true);
}
...

```

`getListCellRendererComponent` 方法获得对这个列表模型的一个引用, 并获得与这个单元值相关联的名字和图标。把名字和图标传送给 `JLabel.setText` 和 `JLabel.setIcon` 方法, `NameAndPictureListCellRenderer` 类扩展 `JLabel`。

```

public Component getListCellRendererComponent (
    JList list,
    Object value,
    int index,
    boolean isSelected,
    boolean cellHasFocus) {
    NameAndPictureListModel model =
        (NameAndPictureListModel) list.getModel ();

    setText (model.getName (value));
    setIcon (model.getIcon (value));
    ...
}

```

如果选取了一个单元, 则设置其背景色和前景色为选取背景色和选取前景色, 它们分别通过调用 `JList.getSelectionBackground()` 和 `JList.getSelectionForeground()` 来获得。

如果这个单元有焦点, 则标签的边框设置为线边框; 如果这个单元没有焦点, 则边框设置为空边框。

`getListCellRendererComponent` 方法返回对绘制器本身的一个引用, 这个绘制器扩展 `JLabel` 类。

```

...
if (isSelected) {
    setForeground (list.getSelectionForeground ());
    setBackground (list.getSelectionBackground ());
}
else {
    setForeground (list.getForeground ());
    setBackground (list.getBackground ());
}

if (cellHasFocus) setBorder (lineBorder);
else              setBorder (emptyBorder);

return this;
}

```

应该注意的是, 通过扩展一个组件类来实现定制绘制器是缺省 Swing 绘制器的实现惯例。然而, 定制绘制器不需要是组件本身。例如, 在前面例子中实现的定制绘制器可能已扩展了 `Object` 并包含了一个 `JLabel` 实例, 它应该从 `getListCellRendererComponent` 方法中返回。

例 17-3 完整地列出了图 17-6 所示小应用程序的代码。

例 17-3 实现一个定制列表单元绘制器

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

```

```

public class Test extends JApplet {
    private String [] names = new String [] {
        "baseball player", "basketball player",
        "beach player", "chef",
        "hockey player", "software developer",
        "construction worker", "martial artist",
        "soccer", "movie star"
    };
    private String [] pics = new String [] {
        "baseball.gif", "basketball.gif",
        "beach_umbrella.gif", "dining.gif",
        "hockey.gif", "mad_hacker.gif",
        "men_at_work.gif", "punch.gif",
        "soccer.gif", "filmstrip.gif"
    };

    public void init () {
        Container contentPane = getContentPane ();
        ListModel model =
            new NameAndPictureListModel (names, pics);

        ListCellRenderer renderer =
            new NameAndPictureListCellRenderer ();

        JList list = new JList (model);
        list.setCellRenderer (renderer);
        list.setVisibleRowCount (5);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (new JScrollPane (list));
    }
}

class NameAndPictureListModel extends DefaultListModel {
    public NameAndPictureListModel (String [] names,
                                    String [] pics) {
        for (int i=0; i < names.length; ++i) {
            addElement (new Object [] {
                names [i], new ImageIcon (pics [i]) });
        }
    }

    public String getName (Object object) {
        Object [] array = (Object []) object;
        return (String) array [0];
    }

    public Icon getIcon (Object object) {
        Object [] array = (Object []) object;
        return (Icon) array [1];
    }
}

class NameAndPictureListCellRenderer extends JLabel
    implements ListCellRenderer {
    private Border
        lineBorder = BorderFactory.createLineBorder (Color.red, 2),
        emptyBorder = BorderFactory.createEmptyBorder (2, 2, 2, 2);

    public NameAndPictureListCellRenderer () {

```



```

        setOpaque (true);
    }
    public Component getListCellRendererComponent (
        JList list,
        Object value,
        int index,
        boolean isSelected,
        boolean cellHasFocus) {
        NameAndPictureListModel model =
            (NameAndPictureListModel) list.getModel ();
        setText (model.getName (value));
        setIcon (model.getIcon (value));
        if (isSelected) {
            setForeground (list.getSelectionForeground ());
            setBackground (list.getSelectionBackground ());
        }
        else {
            setForeground (list.getForeground ());
            setBackground (list.getBackground ());
        }
        if (cellHasFocus) setBorder (lineBorder);
        else setBorder (emptyBorder);
        return this;
    }
}

```

Swing 提示

单元绘制器组件仅用于外观

所有的 Swing 单元绘制器都返回一个被绘制到一个单元中的组件。认识到绘制器组件不能被操纵是很重要的；它们只用于绘制一个单元。

组件总结 17-1 总结了 JList 组件。

组件总结 17-1 JList

模型: AbstractListModel
 UI 代表: javax.swing.plaf.basic.BasicListUI
 绘制器: DefaultListCellRenderer
 编辑器: ——
 激发的事件: PropertyChangeEvent、ListSelectionEvents、ListDataEvents[⊖]
 替换: java.awt.List
 类图: 见图 17-7

JList 类扩展 JComponent，实现 Accessible 接口和 Scrollable 接口，并维护对它的模型、选取模型和绘制器的引用。

⊖ 由列表模型激发。

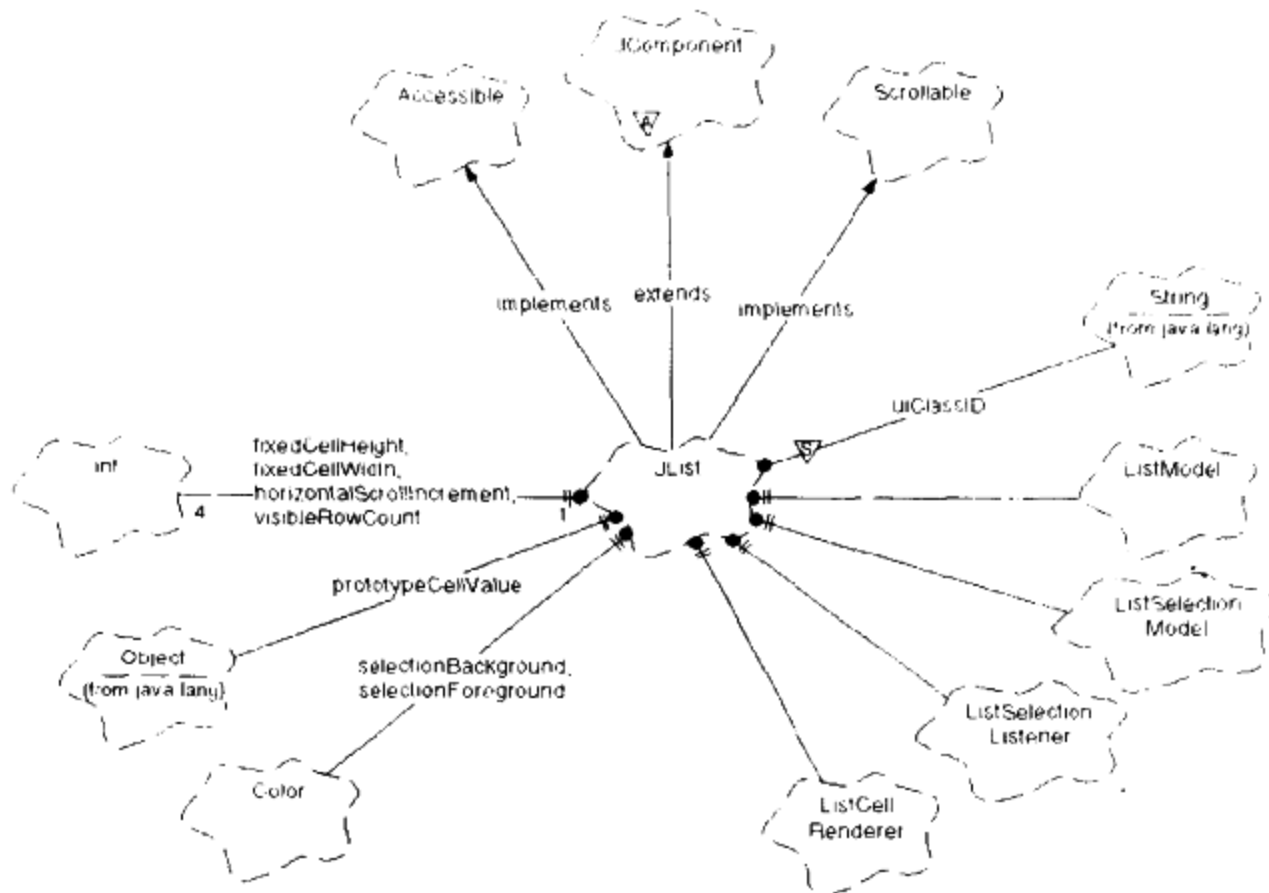


图 17-7 JList 类图

JList 类维护对 ListSelectionListener 的引用，ListSelectionListener 被添加到选取模型中，以便通过把事件发送到已向列表登记的列表选取监听器中，使列表可以对选取作出反应。

JList 类还跟踪它的选取背景色和前景色。这些颜色实际上是由列表的 UI 代表来设置的，但是列表维护这些值。这个列表还维护对 prototypeCellValue（参见例 17-5 “处理列表选取事件”）的引用，还维护对四个 integer 值的引用，这四个引用是：固定单元的宽度和高度、可见行数 and 水平方向滚动的像素增量。

17.3.1 JList 属性

表 17-2 列出了由 JList 维护的属性。

表 17-2 JList 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
anchorSelectionIndex	int	S	G	-1
cellBounds	Rectangle	S	G	---
cellRenderer	ListCellRenderer	B	SG	DefaultListCellRenderer
firstVisibleIndex	int	S	G	---
fixedCellHeight	int	B	SG	-1
fixedCellWidth	int	B	SG	-1
lastVisibleIndex	int	S	G	---
leadSelectionIndex	int	S	G	-1
listData	Object [] / Vector	B	S	null
maxSelectionIndex	int	S	G	integerMAX_VALUE
minSelectionIndex	int	S	G	-1
model	ListModel	B	CSG	AbstractListModel

(续)				
属性名	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
preferredScrollable-ViewportSize	Dimension	S	G	参见下面的讨论
PrototypeCellValue	Object	B	SG	null
scrollableBlockIncrement	int	S	G	参见下面的讨论
scrollableTracks-ViewportHeight	boolean	S	G	参见下面的讨论
scrollableTracks-ViewportWidth	boolean	S	G	参见下面的讨论
scrollableUnitIncrement	int	S	G	参见下面的讨论
selectedIndex	int	B	SG	- 1
selectedIndices	int []	B	SG	int [0]
selectedValue	Object	B	SG	null
selectedValues	Object []	S	G	Object [0]
selectionBackground	Color	B	SG	参见下面的讨论
selectionEmpty	boolean	S	G	true
selectionForeground	Color	B	SG	参见下面的讨论
selectionIndex	boolean	S	SG	- 1
selectionInterval	int, int	B	S	——
selectionMode	int	S	SG	MULTIPLE_ INTERVAL_ SELECTION_
selectionModel	List- Selection- Model	B	SG	DefaultListSelection- Model
valuesAdjusting	boolean	B	SG	false
visibleRowCount	int	B	SG	8

① B = 关联的 (激发 PropertyChangeEvent) / C = 约束的 / I = 索引的 /
S = 简单的 / Ch = 激发 ChangeEvent
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ L&F = 与界面样式有关

anchorSelectionIndex——最后一次的单间隔选取或取消选取的索引。如果只选取了一项，则 anchorSelectionIndex 将与 leadSelectionIndex 相同。如果用户先选取一项，然后在按住 Shift 键的同时单击另一个项，即进行单间隔选取，则 anchorSelectionIndex 将等于第一个选取项的索引。

cellBounds——一个矩形，它代表单间隔选取所占空间。getCellBounds 方法带入两个索引参数，它们分别代表一个列表中的两个单元。这个方法的两索引参数可以相同，此时，这个矩形是一个单元所占的空间。

cellRenderer——一个对象，它绘制列表单元。缺省情况下，JList 的实例配备了一个 DefaultListCellRenderer 实例，它可以显示任何类型的对象，虽然除字符串和图标外，对象显示由它们的 toString 方法返回的字符串。

firstVisibleIndex——列表顶部的第一个可见（整个或部分）单元的索引。如果列表是空

的, 则返回-1。

fixedCellHeight——缺省情况下, 列表单元的高度与列表的最高项高度相同。用 `setFixedCellHeight()` 或一个组件属性, 可以把列表的高度设置为一个固定的像素点值, 这个组件属性是由 `setPrototypeCellValue()` 提供的。

参见 `prototypeCellValue` 和 `fixedCellWidth` 属性。

fixedCellWidth——缺省情况下, 列表单元的宽度与列表的最宽项的宽度相同。用 `setFixedCellWidth()` 或一个组件属性, 可以把列表的宽度设置为一个固定的像素点值, 这个组件属性是由 `setPrototypeCellValue()` 提供的。

参见 `prototypeCellValue` 和 `fixedCellHeight` 属性。

lastVisibleIndex——在列表底部最后可见 (整个或部分) 单元的索引。如果列表是空的, 则返回-1。

参见 `firstVisibleIndex` 属性。

listData——一个只写属性, 它代表在列表中显示的数据, 可以把这个属性指定为一个 `Object` 数组或一个矢量。

maxSelectionIndex——所选单元中最下面的那个选取单元的索引。例如, 如果在列表中选取了索引 2、6、9, 则最大的选取索引是 9。

minSelectionIndex——所选单元中最上面的那个选取单元的索引。例如, 如果在列表中选取了索引 2、6、9, 则最小的选取索引是 2。

model——`ListModel` 接口的一个实现, 它维护在列表单元中显示的数据。

preferredScrollableViewportSize——放在视口中的列表的首选尺寸, 视口通常在 `JScrollPane` 的一个实例中。指定的首选高度是显示由 `visibleRowCount` 属性指定的行数所需的高度, `visibleRowCount` 属性指定的行数可以用 `JList.setVisibleRowCount` 方法来设置, `visibleRowCount` 属性缺省时的设置为 8。

参见 `visibleRowCount`、`scrollableBlockIncrement`、`scrollableTracksViewportWidth` 和 `scrollableTracksViewportHeight` 属性。

prototypeCellValue——为了容纳下列表中最宽的项, 必须允许列表调整其大小, 但我们不这样做, 我们指定一个对象, 把它作为属性单元值。当指定了 `prototypeCellValue` 属性 (缺省时是 `null`) 时, 把列表的 `fixedCellWidth` 和 `fixedCellHeight` 属性分别设置为 `prototypeCellValue` 对象的首选宽度和首选高度。而且, 将列表的字体设置为 `prototypeCellValue` 对象所使用的字体。

参见 `fixedCellWidth` 和 `fixedCellHeight` 属性。

scrollableBlockIncrement——当滚动窗格中的列表滚动了一个块增量时, 表示列表滚动了由 `JList.getScrollableBlockIncrement()` 返回的像素点数。缺省时, 设置了块增量以便滚动列表滚动可见的项数。

由 `Scrollable` 接口来定义 `getScrollableBlockIncrement` 方法。有关 `Scrollable` 接口的更多信息, 请参见 13.3 节 “`Scrollable` 接口”。

参见 `visibleRowCount`、`preferredScrollableViewportSize`、`scrollableTracksViewportWidth` 和 `scrollableTracksViewportHeight` 属性。

scrollableTracksViewportHeight——通过从 `getScrollableTracksViewportHeight` 方法中返回 `true`, 可以强迫可滚动组件的高度与放置该组件的视口的高度相同。有关 `Scrollable` 接口的详细内容, 请参见 13.3 节 “`Scrollable` 接口”。

`JList` 类把它的 `scrollableTracksViewportHeight` 属性设置为 `false`, 以确保它的内容可以被垂直

滚动。

scrollableTracksViewportWidth——通过从 `ScrollableTracksViewportWidth` 方法中返回 `true`，可以强迫可滚动组件的宽度与放置该组件的视口的宽度相同。

`JList` 类把 `scrollableTracksViewportWidth` 属性设置为 `false`，以确保它的内容可以水平地滚动。

scrollableUnitIncrement——`scrollableUnitIncrement` 属性指定当滚动一个单元增量时，列表将滚动的像素数。

缺省情况下，对垂直滚动而言，单元增量是把下一个单元滚动到视图中所需的像素数。

对水平滚动而言，把单元增量设置为列表字体的大小或 1（如果列表有 `null` 字体的话）。

对用固定宽度字体显示文本的列表而言，水平滚动将显示下一个字符。

selectedIndex——`selectedIndex` 属性与 `minIndex` 属性相同。

selectedIndices——一个 `integer` 值数组，这些 `integer` 值代表列表中选的那些单元的索引。

selectedValue——列表中的第一个选取值。

selectedValues——一个 `Object` 引用数组，这些 `Object` 引用代表列表中选的那些项。

selectionBackground——选取单元的背景色。这个颜色由列表的 UI 代表来设置，但是由列表来维护。

selectionEmpty——一个只读的 `boolean` 属性，它指出列表是否有被选取的项。

selectionForeground——选取单元的前景色。这个颜色由列表的 UI 代表来设置，但是由列表来维护。

selectedIndex——列表中第一个选取项的索引。

selectionInterval——选取间隔，它由 `setSelectionInterval (int anchor, int lead)` 或 `addSelectionInterval (int anchor, int lead)` 来指定。

要了解 `setSelectionInterval` 和 `addSelectionInterval` 方法，请参见 17.3.3 节“`JList` 类总结”。

selectionMode——列表的选取模式。选取模式在 17.2 节“列表选取”中介绍。

selectionModel——列表的选取模型。列表在缺省时配备了 `AbstractListModel` 实例，而不是 `DefaultListModel` 实例。在构造列表后，随时都可以调用 `setSelectionModel` 方法来设置选取模型。

valueIsAdjusting——一个 `boolean` 属性，指示上一次在列表项上面拖动鼠标时，是否激发了当前选取事件。如果正在拖动鼠标，则 `ValueIsAdjusting` 属性为 `true`，否则，它就是 `false`。

注意 Swing 1.1 FCS 中的一个 bug 把在按下 `Shift` 或 `Ctrl` 键时单击鼠标错误地报告为调整。

visibleRowCount——当列表包含在一个视口中时显示可见行数。如果列表没有包含在一个视口中，则设置 `visibleRowCount` 属性没有效果。

17.3.2 `JList` 事件

`JList` 类激发两种类型的事件：`PropertyChangeEvents` 和 `ListSelectionEvents`。另外，当修改列表数据时，列表的模型会激发 `ListDataEvents`。

1. 列表选取事件

实现 `ListSelectionListener` 接口的对象处理列表选取事件，接口总结 17-4 总结了 `ListSelectionListener`。

接口总结 17-4 `ListSelectionListener`

```
public abstract void valueChanged (ListSelectionEvent)
```

当列表的选取变化时，就为列表选取监听器调用 `valueChanged` 方法。这个方法带入对 `ListSelectionEvent` 的一个引用。

类总结 17-4 总结了 `ListSelectionEvent` 类。

类总结 17-4 `ListSelectionEvent`

扩展：`java.util.EventObject`

1. 构造方法

`public ListSelectionEvent (Object source, int firstIndex, int lastIndex, boolean valueIsAdjusting)`

`ListSelectionEvent` 由对事件源的一个引用、变化的第一个索引和最后一个索引以及一个 `boolean` 变量来构造，其中的 `boolean` 变量指示这个事件是否是一系列事件之一。当鼠标正在列表项上拖动时，`valueIsAdjusting` 参数是 `true`，当释放鼠标按钮时，`valueIsAdjusting` 参数是 `false`。

2. 方法

`public int getFirstIndex ()`
`public int getLastIndex ()`
`public boolean getValueIsAdjusting ()`
`public String toString ()`

由 `ListSelectionEvent` 类实现的这些方法是传送给 `ListSelectionEvent` 构造方法的那些值的访问方法。`toString` 方法返回一个代表事件的字符串，这个方法对事件日志和调试是很有用的。

图 17-8 所示的小应用程序说明了具有调整值的选取事件。当光标在前三项上拖动时，将激发列表选取事件，并且调整了选取事件的值。当释放鼠标按钮后，将产生一个选取事件，这个选取事件的值不调整，结果如右下角的图片所示。

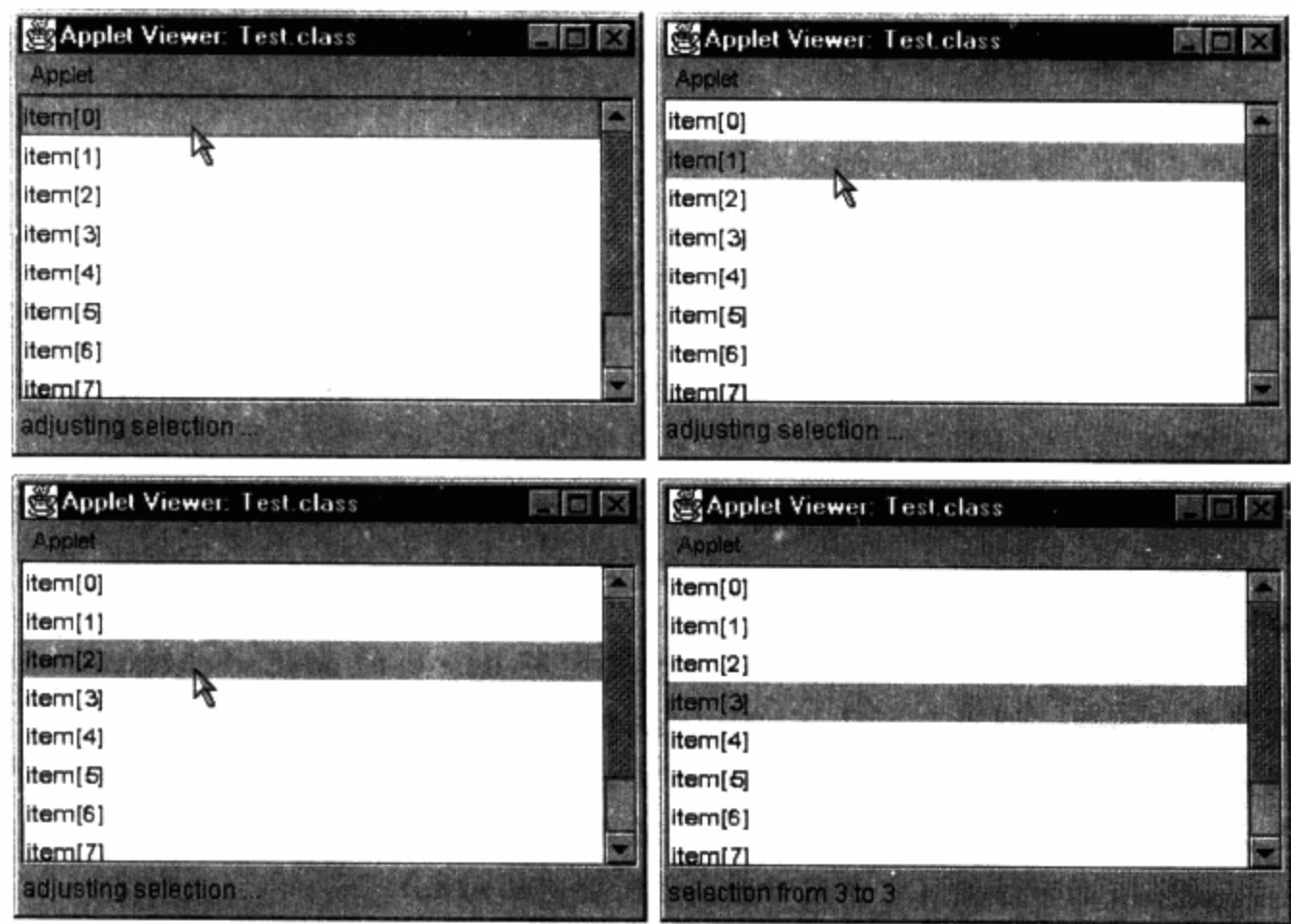


图 17-8 列表选取的调整值

例 17-4 列出了图 17-8 所示小应用程序的代码。

例 17-4 检测列表选取的调整值

```
import java.awt.* ;
import javax.swing.* ;
import javax.swing.event.* ;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane () ;

        String [] items = { "item [0]", "item [1]", "item [2]",
                            "item [3]", "item [4]", "item [5]",
                            "item [6]", "item [7]",
                            "item [8]", "item [9]" } ;

        JList list = new JList (items) ;

        contentPane.add (new JScrollPane (list),
                        BorderLayout.CENTER) ;

        list.addListSelectionListener (
            new ListSelectionListener () {
                public void valueChanged (ListSelectionEvent e) {
                    String s ;

                    if (e.getValueAdjusting ()) {
                        s = "adjusting selection ..." ;
                    }
                    else {
                        s = "selection from " + e.getFirstIndex () +
                            " to " + e.getLastIndex () ;
                    }

                    showStatus (s) ;
                }
            }
        ) ;
    }
}
```

这个小应用程序创建一个包裹在滚动窗格中的列表，并把一个列表选取监听器添加到这个列表中。根据与事件相关联的值是否在调整，这个监听器在这个小应用程序的内容窗格中显示一个相应的字符串。

Swing 提示

对列表选取作出反应

有两种登记列表选取事件监听器的办法：通过调用 JList 类的 addListSelectionListener 方法来直接向列表登记，或通过调用 ListSelectionModel 类的同名方法来向列表的选取监听器登记。列表选取模型是可以通过 JList.getSelectionModel 方法访问的，这使得第二种办法是可行的。

向列表登记与向列表的模型登记之间的唯一差别是传送给 ListSelectionListener.valueChanged 方法的 ListSelectionEvent 的事件源（它是监听器要进行登记的对象）。

图 17-9 示出的小应用程序包含一个列表和一个面板。这个面板包含一个用于选取列表选

取模式的组合框和一些显示这个列表当前选取属性的标签。

表 17-3 简要重述了选取属性和它们的含义。

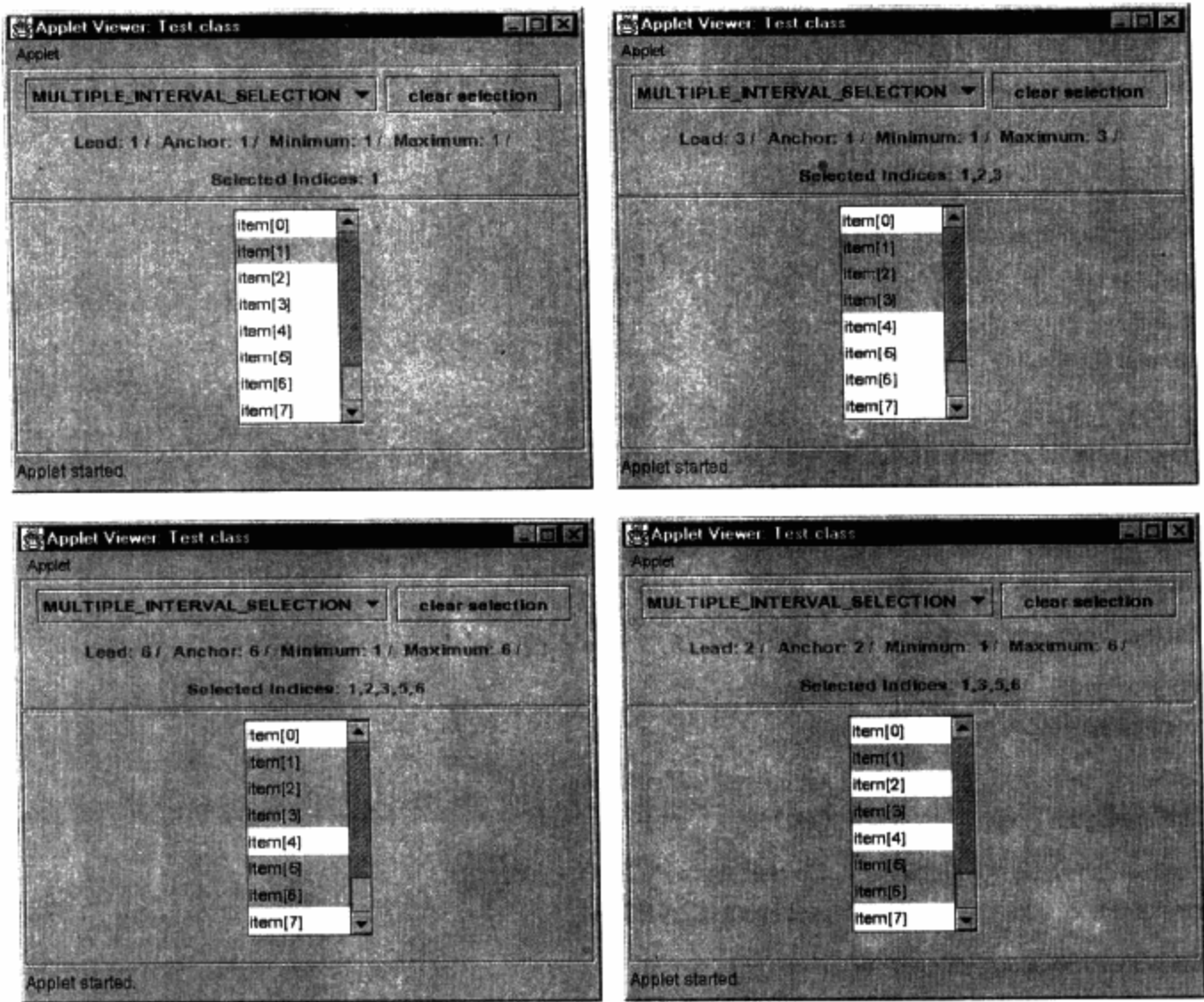


图 17-9 列表选取

图 17-9 示出的四个图片显示列表的选取模式设置为多间隔选取时的样子。每个图片显示不同的选取项集和相应的选取属性值。这些图片是用下面的鼠标单击序列产生的。

左上角的图片：

在 item [1] 上单击。

右上角的图片：

按下 Shift 同时在 item [3] 上单击。

左下角的图片：

按下 Ctrl 同时在 item [5] 上单击。

按下 Ctrl 同时在 item [6] 上单击。

右下角的图片：

按下 Ctrl 键的同时在 item [2] 上单击。

表 17-3 JList 选取属性

属性	含义
anchor	最后一次锚定一个间隔选取的索引
lead	最后一次指定的选取间隔的最后索引
maximum	选取项中的最大索引
minimum	选取项中的最小索引

未修饰的鼠标单击清除当前的选取（如果有的话）并选取鼠标单击的项。按下 Shift 键的同时单击鼠标将把单击项的索引设置为 lead 索引，而按下 Ctrl 键的同时单击鼠标则把单击的项添加（如果没有选取该项）到列表的选取项集中，或从选取项集中删除单击的项（如果已选取了该项）。

这个小应用程序创建一个列表和一个控制面板，这个控制面板包含一个组合框、一个按钮和一些标签。列表包裹在一个滚动窗格中，并且控制面板和滚动窗格都添加到这个小应用程序

的内容窗格中。添加一个列表选取监听器到这个列表中，以便在修改列表的选取状态时更新这个控制面板。

这个小应用程序还调用 `JList.setPrototypeCellValue (Object)`，它把列表的宽度设置为指定对象的宽度。缺省时，`JList` 的实例在水平方向上调整大小，以便列表的宽度与它的最宽项的宽度相同。利用 `setPrototypeCellValue` 方法，可以显式地控制列表的宽度，并且能够提高绘制列表的性能。

```
public class Test extends JApplet {
    private ControlPanel controlPanel;

    public void init () {
        Container contentPane = getContentPane ();
        JPanel listPanel = new JPanel ();

        String [] items = { "item [0]", "item [1]", "item [2]",
                           "item [3]", "item [4]", "item [5]",
                           "item [6]", "item [7]",
                           "item [8]", "item [9]" };

        JList list = new JList (items);

        list.setPrototypeCellValue ("MMMMMMMM");

        controlPanel = new ControlPanel (list);
        controlPanel.update ();

        listPanel.setBorder (BorderFactory.createEtchedBorder ());
        listPanel.add (new JScrollPane (list));

        contentPane.add (controlPanel, BorderLayout.NORTH);
        contentPane.add (listPanel, BorderLayout.CENTER);

        list.addListSelectionListener (
            new ListSelectionListener () {
                public void valueChanged (ListSelectionEvent e) {
                    controlPanel.update ();
                }
            });
    }
}
```

`ControlPanel` 类扩展 `JPanel` 类并创建组合框、按钮和标签，它们被添加到控制面板中的三个面板中。

用 `initializeSelectionMode` 方法（如下所列）来将这个组合框实例化，一个在这个组合框中添加的项监听器在选取一个项时将设置选取模式。

添加到清除按钮中的动作监听器在激活这个按钮时将调用 `JList.clearSelection ()`。

```
class ControlPanel extends JPanel {
    private JComboBox mode = new JComboBox ();
    private JButton clear = new JButton ("clear selection");

    private String single = "SINGLE_SELECTION",
        singleInterval = "SINGLE_INTERVAL_SELECTION",
        multipleInterval = "MULTIPLE_INTERVAL_SELECTION";

    private JLabel leadLabel = new JLabel (),
        anchorLabel = new JLabel (),
        minLabel = new JLabel (),
        maxLabel = new JLabel ();
```

```

        selIndicesLabel = new JLabel ();

private JList list;

public ControlPanel (JList l) {
    //create three panels...

    this.list = l;

    // add combo box , button, and labels to the panels
    //created above...

    mode.addItem (single);
    mode.addItem (singleInterval);
    mode.addItem (multipleInterval);

    initializeSelectionMode ();

    ...
    mode.addItemListener (new ItemListener () {
        public void itemStateChanged (ItemEvent e) {
            if (e.getStateChange () == ItemEvent.SELECTED)
                setSelectionMode ((String) e.getItem ());
        }
    });
    clear.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            list.clearSelection ();
        }
    });
}
}

```

initializeSelectionMode 方法在 ControlPanel 构造方法中调用，而且它根据这个列表的初始选取模式来设置组合框中的选取项。

```

...
private void initializeSelectionMode () {
    int m = list.getSelectionMode ();

    switch (m) {
        case ListSelectionModel.SINGLE_SELECTION:
            mode.setSelectedItem (single);
            break;
        case ListSelectionModel.SINGLE_INTERVAL_SELECTION:
            mode.setSelectedItem (singleInterval);
            break;
        case ListSelectionModel.MULTIPLE_INTERVAL_SELECTION:
            mode.setSelectedItem (multipleInterval);
            break;
    }
}
...

```

注意，initializeSelectionMode 不是简单地去完成初始化这个组合框的任务，因为 JList 实例的缺省选取模式是已知的(MULTIPLE_SELECTION_MODE)，因此，在不必询问列表的选取模式的情况下也能够初始化这个组合框。但是，这将依赖于 JList 类的实现细节而不是依赖于 JList API[⊖]。

⊖ 好的面向对象设计与类的实现细节无关。

实际上, 如果将来修改了 JList 的实例的缺省选取模式, 则 initializeSelectionMode 方法仍能正常的工作。

当从组合框中选取了一个项时, 就调用 setSelectionMode 方法, 并把选取的字符串作为 setSelectionMode 方法的参数。setSelectionMode 方法根据传送给它的字符串来设置列表的选取模式。

```
...
private void setSelectionMode (String s) {
    if (s.equals ("SINGLE_SELECTION")) {
        list.setSelectionMode (
            ListSelectionMode.SINGLE_SELECTION);
    }
    else if (s.equals ("SINGLE_INTERVAL_SELECTION")) {
        list.setSelectionMode (
            ListSelectionMode.SINGLE_INTERVAL_SELECTION);
    }
    else if (s.equals ("MULTIPLE_INTERVAL_SELECTION")) {
        list.setSelectionMode (
            ListSelectionMode.MULTIPLE_INTERVAL_SELECTION);
    }
}
...
```

当修改列表的选取状态时, 将调用 update 方法。从这个列表中获得 lead、min、max 和 anchor 选取属性, 并更新控制面板中的标签。在把这些标签更新后, 就调用用于控制面板的 validate (), 使这个面板被布局。当修改标签显示的字符串长度时, 必须调用 validate ()。

```
...
public void update () {
    int lead = list.getLeadSelectionIndex ();
    min = list.getMinSelectionIndex ();
    max = list.getMaxSelectionIndex ();
    anchor = list.getAnchorSelectionIndex ();

    leadLabel.setText (Integer.toString (lead) + " / ");
    anchorLabel.setText (Integer.toString (anchor) + " / ");
    minLabel.setText (Integer.toString (min) + " / ");
    maxLabel.setText (Integer.toString (max) + " / ");

    int [] selected = list.getSelectedIndices ();
    String s = new String ();

    for (int i = 0; i < selected.length; ++i) {
        s += Integer.toString (selected [i]);

        if (i < selected.length-1)
            s += ",";
    }
    selIndicesLabel.setText (s);
    validate ();
}
...
```

例 17-5 完整地列出了图 17-9 所示小应用程序的代码。

例 17-5 处理列表选取事件

```
import java.awt.*;
```

```

import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.event. * ;

public class Test extends JApplet {
    private ControlPanel controlPanel;

    public void init () {
        Container contentPane = getContentPane ();
        JPanel listPanel = new JPanel ();

        String [] items = { "item [0]", "item [1]", "item [2]",
                            "item [3]", "item [4]", "item [5]",
                            "item [6]", "item [7]",
                            "item [8]", "item [9]" };

        JList list = new JList (items);

        list.setPrototypeCellValue ("MMMMMMMM");

        controlPanel = new ControlPanel (list);
        controlPanel.update ();

        listPanel.setBorder (BorderFactory.createEtchedBorder ());
        listPanel.add (new JScrollPane (list));

        contentPane.add (controlPanel, BorderLayout.NORTH);
        contentPane.add (listPanel, BorderLayout.CENTER);

        list.addListSelectionListener (
            new ListSelectionListener () {
                public void valueChanged (ListSelectionEvent e) {
                    controlPanel.update ();
                }
            }
        );
    }

    class ControlPanel extends JPanel {
        private JComboBox mode = new JComboBox ();
        private JButton clear = new JButton ("clear selection");

        private String single = "SINGLE _ SELECTION",
            singleInterval = "SINGLE _ INTERVAL _ SELECTION",
            multipleInterval = "MULTIPLE _ INTERVAL _ SELECTION";

        private JLabel leadLabel = new JLabel (),
            anchorLabel = new JLabel (),
            minLabel = new JLabel (),
            maxLabel = new JLabel (),
            selIndicesLabel = new JLabel ();

        private JList list;

        public ControlPanel (JList l) {
            JPanel top = new JPanel (),
                mid = new JPanel (),
                bottom = new JPanel ();

            this.list = l;

            setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
            setBorder (BorderFactory.createEtchedBorder ());
            top.add (mode);

```

```

top.add (clear);

mid.add (new JLabel ("Lead:")); mid.add (leadLabel);
mid.add (new JLabel ("Anchor:")); mid.add (anchorLabel);
mid.add (new JLabel ("Minimum:")); mid.add (minLabel);
mid.add (new JLabel ("Maximum:")); mid.add (maxLabel);

add (top);
add (mid);
add (bottom);

mode.addItem (single);
mode.addItem (singleInterval);
mode.addItem (multipleInterval);
initializeSelectionMode ();

bottom.add (new JLabel ("Selected Indices:"));
bottom.add (selIndicesLabel);

mode.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent e) {
        if (e.getStateChange () == ItemEvent.SELECTED)
            setSelectionMode ((String) e.getItem ());
    }
});

clear.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        list.clearSelection ();
    }
});

}

public void update () {
    int lead = list.getLeadSelectionIndex ();
    min = list.getMinSelectionIndex ();
    max = list.getMaxSelectionIndex ();
    anchor = list.getAnchorSelectionIndex ();

    leadLabel.setText (Integer.toString (lead) + " / ");
    anchorLabel.setText (Integer.toString (anchor) + " / ");
    minLabel.setText (Integer.toString (min) + " / ");
    maxLabel.setText (Integer.toString (max) + " / ");

    int [] selected = list.getSelectedIndices ();
    String s = new String ();

    for (int i = 0; i < selected.length; ++i) {
        s += Integer.toString (selected [i]);

        if (i < selected.length-1)
            s += ",";
    }

    selIndicesLabel.setText (s);
    validate ();
}

private void initializeSelectionMode () {
    int m = list.getSelectionMode ();

    switch (m) {
        case ListSelectionModel.SINGLE_SELECTION:
            mode.setSelectedItem (single);

```

```

        break;
    case ListSelectionMode.SINGLE_INTERVAL_SELECTION:
        mode.setSelectedItem (singleInterval);
        break;
    case ListSelectionMode.MULTIPLE_INTERVAL_SELECTION:
        mode.setSelectedItem (multipleInterval);
        break;
    }
}

private void setSelectionMode (String s) {
    if (s.equals ("SINGLE_SELECTION")) {
        list.setSelectionMode (
            ListSelectionMode.SINGLE_SELECTION);
    }
    else if (s.equals ("SINGLE_INTERVAL_SELECTION")) {
        list.setSelectionMode (
            ListSelectionMode.SINGLE_INTERVAL_SELECTION);
    }
    else if (s.equals ("MULTIPLE_INTERVAL_SELECTION")) {
        list.setSelectionMode (
            ListSelectionMode.MULTIPLE_INTERVAL_SELECTION);
    }
}
}

```

2. 列表数据事件

当修改一个列表的数据时，列表模型激发将列表数据事件。与大多数 Swing 组件不同，JList 类不监听它的模型来把事件发送给已向这个列表登记的监听器。如果需要列表数据事件的通知，则必须直接向列表模型登记监听器。

接口总结 17-5 总结 ListDataListener 接口。

接口总结 17-5 ListDataListener

```

public abstract void contentsChanged (ListDataEvent)
public abstract void intervalAdded (ListDataEvent)
public abstract void intervalRemoved (ListDataEvent)

```

无论何时修改与列表相关联的数据，都会调用由 ListDataListener 类定义的三个方法中的一个方法。当向列表模型添加项间隔或从列表模型中删除项间隔时，则会分别调用 intervalAdded 方法和 intervalRemoved 方法。如果对列表数据进行了一种比简单地添加或删除一个间隔更复杂的修改，则调用 contentsChanged 方法。

列表数据监听器最有趣的方面是跟踪传送给由 ListDataListener 接口定义的方法的事件。类总结 17-5 总结了 ListDataEvent 类。

类总结 17-5 ListDataEvent

扩展：java.util.EventObject

1. 常量

```

public static final int CONTENTS_CHANGED
public static final int INTERVAL_ADDED
public static final int INTERVAL_REMOVED

```


上面列出的常量定义列表数据事件的类型。

2. 构造方法

```
public ListDataEvent (Object source, int type, int index0, int index1)
```

ListDataEvent 以事件源、事件类型（由上面所列的这些 static final 常量之一来定义）和被修改的项范围中的开始和结束索引为参数来构造。

3. 方法

```
public int getIndex0 ()
public int getIndex1 ()
public int getType ()
```

图 17-10 示出的小应用程序包含一个列表和两个按钮，其中一个按钮从列表中删除选取项，另一个按钮重新填充这个列表。这个小应用程序把一个列表数据监听器添加到列表中，无论何时从列表中删除一个项间隔，这个列表都显示一个对话框。

图 17-10 示出的左上图显示的是这个小应用程序开始时的样子。在右上图中，选取了列表中的第三项（索引为 2）。左下图中示出了激活 remove selected items 按钮的结果。示出的对话框指示删除的项和保留的项数。右下图显示项已经被删除后列表的样子。

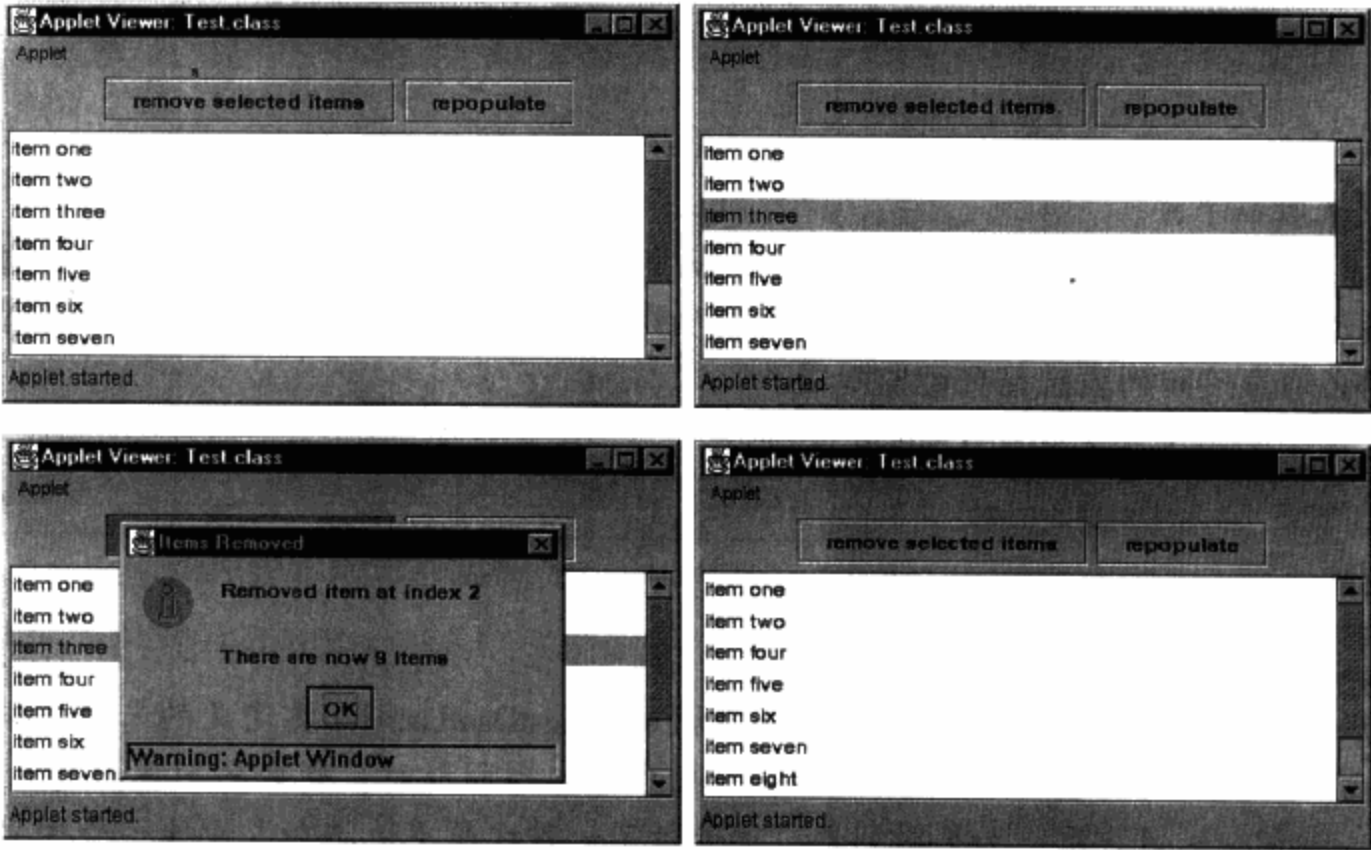


图 17-10 处理列表数据事件

这个小应用程序创建一个包裹在滚动窗格中的列表，并把滚动窗格添加到这个小应用程序的内容窗格中。然后，再用一些字符串来填充这个列表。

```
public class Test extends JApplet {
    private JList list = new JList ();

    private String [] items = {
        "item one", "item two", "item three",
        "item four", "item five", "item six",
        "item seven", "item eight",
        "item nine", "item ten"
    };
}
```

```

public void init () {
    Container contentPane = getContentPane ();
    JPanel controlPanel = new ControlPanel (this, list);
    contentPane.add (controlPanel, BorderLayout.NORTH);
    contentPane.add (new JScrollPane (list),
                    BorderLayout.CENTER);
    populateList ();
}

```

populateList 方法创建了一个 DefaultListModel 实例，并添加项数组中的字符串，然后，把这个模型指定为列表模型。这个模型配备了一个 ListDataListener 实例，此实例对删除间隔作出反应，即显示一个消息对话框。

```

public void populateList () {
    final DefaultListModel model = new DefaultListModel ();
    for (int i=0; i < items.length; ++i)
        model.addElement (items [i]);
    list.setModel (model);
    if (list.isShowing ())
        list.revalidate ();
    model.addListDataListener (new ListDataListener () {
        public void contentsChanged (ListDataEvent e) {
            showStatus ("contents changed");
        }
        public void intervalRemoved (ListDataEvent e) {
            Object [] message = new Object [] {
                "Removed item at index " + e.getIndex0 (),
                " ",
                "There are now " + model.getSize () + " items"
            };
            JOptionPane.showMessageDialog (Test.this,
                message,
                "Items Removed", // title
                JOptionPane.INFORMATION_MESSAGE); // type
        }
        public void intervalAdded (ListDataEvent e) {
            showStatus ("interval added");
        }
    });
}

```

ControlPanel 扩展 JPanel 并且包含这个小应用程序的 remove selected items 按钮和 repopulate 按钮。这些按钮都配备了动作监听器，这些监听器从模型中删除元素并重新填充这个列表。

```

class ControlPanel extends JPanel {
    JButton remove = new JButton ("remove selected items");
    JButton repopulate = new JButton ("repopulate");
    public ControlPanel (final Test applet, final JList list) {
        add (remove);
        add (repopulate);
        remove.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {

```

```

        int [] selected = list.getSelectedIndices ();
        DefaultListModel model =
            (DefaultListModel) list.getModel ();
        for (int i=0; i < selected.length; ++i) {
            model.removeElementAt (selected [i] - i);
        }
    }
    );
    repopulate.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            applet.populateList ();
        }
    });
}
}

```

例 17-6 完整地列出了图 17-10 所示小应用程序的代码。

例 17-6 处理列表数据事件

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.event. * ;

public class Test extends JApplet {
    private JList list = new JList ();
    private String [] items = {
        "item one", "item two", "item three",
        "item four", "item five", "item six",
        "item seven", "item eight",
        "item nine", "item ten"
    };

    public void init () {
        Container contentPane = getContentPane ();
        JPanel controlPanel = new ControlPanel (this, list);
        contentPane.add (controlPanel, BorderLayout.NORTH);
        contentPane.add (new JScrollPane (list),
            BorderLayout.CENTER);
        populateList ();
    }

    public void populateList () {
        final DefaultListModel model = new DefaultListModel ();
        for (int i=0; i < items.length; ++i)
            model.addElement (items [i]);

        list.setModel (model);
        if (list.isShowing ())
            list.revalidate ();

        model.addListDataListener (new ListDataListener () {
            public void contentsChanged (ListDataEvent e) {
                showStatus ("contents changed");
            }
        });
    }
}

```

```

    public void intervalRemoved (ListDataEvent e) {
        Object [] message = new Object [] {
            "Removed item at index " + e.getIndex0 (),
            "",
            "There are now " + model.getSize () + " items"
        };
        JOptionPane.showMessageDialog (Test.this,
            message,
            "Items Removed", // title
            JOptionPane.INFORMATION_MESSAGE); // type
    }

    public void intervalAdded (ListDataEvent e) {
        showStatus ("interval added");
    }
}

class ControlPanel extends JPanel {
    JButton remove = new JButton ("remove selected items");
    JButton repopulate = new JButton ("repopulate");

    public ControlPanel (final Test applet, final JList list) {
        add (remove);
        add (repopulate);

        remove.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int [] selected = list.getSelectedIndices ();
                DefaultListModel model =
                    (DefaultListModel) list.getModel ();

                for (int i=0; i < selected.length; ++i) {
                    model.removeElementAt (selected [i] - i);
                }
            }
        });

        repopulate.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                applet.populateList ();
            }
        });
    }
}

```

3. 处理鼠标双击和三击

本节讨论对列表中鼠标的双击和三击的处理。这种讨论给我们提供了一个机会，以便说明如何把发生鼠标点击的点转换成一个索引，以及如何检测一个鼠标点击是双击还是三击。

图 17-11 示出的小应用程序包含一个包裹在滚动窗格中的列表。当在列表上双击或三击时，这个小应用程序的状态条相应地更新。

这个小应用程序以一个字符串数组为参数来创建一个 JList 实例，并把这个列表放在滚动面板中。然后，把这个滚动面板添加到这个列表的内容窗格中。

```

public class Test extends JApplet {

```

```

public void init () {
    Container contentPane = getContentPane ();

    object [] items = { "item one", "item two", "item three",
        "item four", "item five", "item six",
        "item seven", "item eight",
        "item nine", "item ten" };

    JList list = new JList (items);

    contentPane.setLayout (new FlowLayout ());
    contentPane.add (new JScrollPane (list));
    ...

```

添加一个鼠标监听器到列表中以便更新这个小应用程序的状态条。这个列表是事件源，因此可以通过调用 `EventObject.getSource()` 来获得对列表的一个引用。在这个列表中，可访问列表的模型，而且通过调用 `JList.locationToIndex()` 还能获得单击项的索引。

一旦获得了这个模型并知道了索引值，则通过调用 `AbstractListModel.elementAt()` 来获得与此项相关联的字符串。用这个项字符串和单击类型来构造一个字符串。`getClickCount` 方法检测这个点击是单击、双击还是三击。

```

...
list.addMouseListener (new MouseAdapter () {
    public void mouseClicked (MouseEvent e) {
        JList theList = (JList) e.getSource ();
        ListModel model = theList.getModel ();

        int index = theList.locationToIndex (e.getPoint ());
        String itemString =
            (String) model.elementAt (index);

        String s = new String (" for " +
            model.elementAt (index));

        switch (e.getClickCount ()) {
            case 1:
                showStatus ("Single Click" + s);
                break;
            case 2:
                showStatus ("Double Click" + s);
                break;
            case 3:
                showStatus ("Triple Click" + s);
                break;
        }
    }
});
}

```

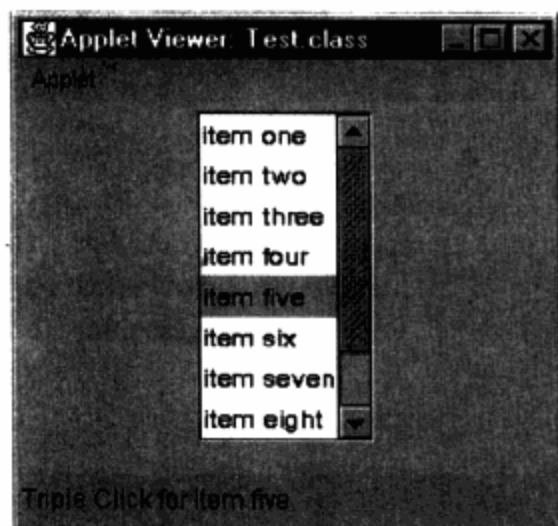


图 17-11 检测列表中的双击和三击

例 17-7 完整地列出了图 17-11 所示的小应用程序的代码。

例 17-7 处理鼠标双击和三击

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;

```

```

import javax.swing.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();

        object [] items = { "item one", "item two", "item three",
                            "item four", "item five", "item six",
                            "item seven", "item eight",
                            "item nine", "item ten" };

        JList list = new JList (items);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (new JScrollPane (list));

        list.addMouseListener (new MouseAdapter () {
            public void mouseClicked (MouseEvent e) {
                JList theList = (JList) e.getSource ();
                ListModel model = theList.getModel ();

                int index = theList.locationToIndex (e.getPoint ());
                String itemString =
                    (String) model.getElementAt (index);

                String s = new String (" for " +
                    model.getElementAt (index));

                switch (e.getClickCount ()) {
                    case 1:
                        showStatus ("Single Click" + s);
                        break;
                    case 2:
                        showStatus ("Double Click" + s);
                        break;
                    case 3:
                        showStatus ("Triple Click" + s);
                        break;
                }
            }
        });
    }
}

```

17.3.3 JList 类总结

类总结 17-6 列出了 JList 的 public 和 protected 的变量和方法。

类总结 17-6 JList

扩展: JComponent

实现: Scrollable、javax.accessibility.Accessible

1. 构造方法

```

public JList ()
public JList (ListModel)
public JList (Object [])
public JList (Vector)

```

JList 类可以不带数据构造, 也可以带指定为一个 Object 数组、矢量或列表模型的数据来构

造。上面列出的第二个构造方法以 `DefaultListModel` 的实例为参数, `DefaultListModel` 提供了插入和删除项的方法。

用一个 `Object` 数组或一个矢量构造的 `JList` 实例配备了一个简单的、扩展了 `AbstractListModel` 的列表模型。

2. 方法

(1) 列表选取事件

```
public void addListSelectionListener (ListSelectionListener)
public void removeListSelectionListener (ListSelectionListener)
protected void fireSelectionValueChanged (int firstIndex, int lastIndex, boolean isAdjusting)
```

上面所列的前两个方法向 `JList` 实例登记列表选取监听器。当第一次为列表调用 `addListSelectionListener` 时, 一个选取监听器被初始化并向这个列表的选取模型进行登记。`addListSelectionListener` 方法还向列表的选取监听器登记指定的监听器。

当列表的选取模型激发一个列表选取事件时, 列表的选取监听器把事件发送给列表自己的选取监听器。这一切的结果是传送给 `addListSelectionListener` 的监听器都收到了带事件源的选取变化通知, 事件源就是这个列表。已向列表选取模型登记的监听器将收到相同的事件通知, 但事件源将是选取模型而不是列表。

`protected fireSelectionValueChanged` 方法把选取事件发送给已向列表登记的列表选取监听器。如果通知监听器的方式需要改变或改进, 则可以重载这个方法。

(2) 列表选取

```
protected ListSelectionModel createSelectionModel ()
public void addSelectionInterval (int, int)
public void removeSelectionInterval (int, int)
public void clearSelection ()
public int getAnchorSelectionIndex ()
public int getLeadSelectionIndex ()
public int getMaxSelectionIndex ()
public int getMinSelectionIndex ()
public int getSelectionIndex ()
public int [] getSelectionIndices ()
public Object getSelectionValue ()
public Object [] getSelectionValues ()
public ListSelectionModel getSelectionModel ()
public int getSelectionModel ()
public boolean isSelectedIndex (int)
public boolean isSelectionEmpty ()
public void setSelectedIndex (int)
public void setSelectedIndices (int [])
public void setSelectedValue (Object, boolean)
public void setSelectionInterval (int, int)
public void setSelectionMode (int)
public void setSelectionModel (ListSelectionModel)
```

上面所列的所有方法都与列表选取有关。许多方法把它们的功能实现简单、直接地交给列表的选取模型。例如, `getAnchorSelectionIndex` 方法是以如下方式实现的:

```
// From JList.java
public int getAnchorSelectionIndex () {
```



```
return getSelectionModel().getAnchorSelectionIndex();
```

`createSelectionModel()` 创建一个选取模型。提供这个方法是为了使 `JList` 的扩展可以重载它来安装一个定制选取模型。应该注意的是, 可以通过简单地调用 `JList.setSelectionModel()` 来安装一个列表的定制选取模型。扩展 `JList` 和重载 `createSelectionModel` 通常是为这样一些定制列表组件保留的, 这些定制列表组件是与它们的选取模型紧密地结合在一起的。

可以用 `setSelectionMode` 方法设置列表选取模式, `setSelectionMode` 方法以下面的 `integer` 常量之一为参数:

- `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`
- `ListSelectionModel.SINGLE_INTERVAL_SELECTION`
- `ListSelectionModel.SINGLE_SELECTION`

如果选取模式是 `SINGLE_INTERVAL_SELECTION` 或 `MULTIPLE_INTERVAL_SELECTION`, 则 `setSelectionInterval` 方法清除当前的选取并选取指定的间隔。如果选取模式是 `SINGLE_SELECTION`, 则这个方法没有效果。

`addSelectionInterval` 方法把指定间隔添加到当前选取项中, 如果选取模式是 `MULTIPLE_INTERVAL_SELECTION`。如果选取模式是 `SINGLE_INTERVAL_SELECTION`, 这个方法清除当前的选取, 并选取指定的间隔。如果选取模式是 `SINGLE_SELECTION`, 这个方法没有效果。

(3) Scrollable 方法

```
public Dimension getPreferredSize()
public int getScrollableBlockIncrement (Rectangle, int, int)
public boolean getScrollableTracksViewportHeight ()
public boolean getScrollableTracksViewportWidth ()
public int getScrollableUnitIncrement (Rectangle, int, int)
```

上面所列的这些方法由 `Scrollable` 接口来定义。

`getPreferredSize` 方法为包含在一个视口中的列表返回首选视口。列表希望它们的视口与它们最宽项的宽度相同, 视口的高与可见行数等高, 可见行数则由 `JList.setVisibleRowCount()` 指定。

`getScrollableBlockIncrement` 返回垂直滚动列表当前可见部分的高度, 或水平滚动列表当前可见部分的宽度。这种运算规则导致块滚动, 它滚动列表的可见部分。

对垂直滚动而言, `getScrollableUnitIncrement()` 返回把下一个单元滚动到视图所需的像素数。对水平滚动而言, 这个方法返回的值是列表的字体大小或 1。

`getScrollableTracksViewportWidth` 和 `getScrollableTracksViewportHeight` 方法返回 `boolean` 值, 指示列表是否希望把它的宽度和高度保持与它的视口大小相同, 这又称作跟踪视口大小。比视口大的列表不跟踪视口大小, 因此, 这个方法返回 `false`。比视口小的列表跟踪视口的大小, 即这个方法返回 `true`。

(4) 属性单元和固定宽度和高度

```
public int getFixedCellHeight ()
public int getFixedCellWidth ()
public Object getPrototypeCellValue ()
public void setFixedCellHeight (int)
public void setFixedCellWidth (int)
public void setPrototypeCellValue (Object)
```

缺省情况下, 列表把它们的单元宽度调整为与列表最宽项的宽度相同, 把它们的单元高度

调整为与每个单元绘制器组件的总高度相同。用一个 integer 存储列表最宽项的宽度，用一个 integers 数组来代表单元高度。当修改列表数据时，这些值都必须重新计算。

另一个调整列表单元大小的方法是使用 `setFixedCell...()` 方法，用这些方法来指定一个固定单元的宽度和高度。如果明确地指定了固定单元的宽度或高度，则固定单元的宽度或高度不会像上面所述的那样计算。

除了可以用这些方法来设置固定单元的宽度和高度外，还可以通过给 `setPrototypeCellValue()` 传送一个对象来间接地指定固定单元的宽度和高度。这个对象（指定为属性单元值）的首选大小设置列表固定单元的宽度和高度。

(5) 属性访问方法

```
public Rectangle getCellBounds (int index1, int index2)
public ListCellRenderer getCellRenderer ()
public int getFirstVisibleIndex ()
public int getLastVisibleIndex ()
public ListModel getModel ()
public Color getSelectionBackground ()
public Color getSelectionForeground ()
public boolean getValuesAdjusting ()
public int getVisibleRowCount ()

public void setCellRenderer (ListCellRenderer)
public void setListData (Object [])
public void setListData (Vector)
public void setModel (ListModel)
public void setSelectionBackground (Color)
public void setSelectionForeground (Color)
public void setValuesAdjusting (boolean)
public void setVisibleRowCount (int)
```

上面所列的这些方法是 JList 属性的访问方法，JList 属性在 17.3.1 节“JList 属性”中介绍。

(6) 项索引和项定位

```
public void ensureIndexIsVisible (int)
public Point indexToLocation (int)
public int locationToIndex (Point)
```

只有当列表被包含在一个视口中时，`ensureIndexIsVisible` 方法才能发挥作用。`ensureIndexIsVisible()` 方法调用 `scrollRectToVisible()` 方法来把与指定的索引相关联的单元滚动到视图中。有关视口和 `scrollRectToVisible` 方法的更多信息，请参见 13.1.2 节“使用 `scrollRectToVisible` 方法”。

如果为一个刚被添加到列表中的项调用 `ensureIndexIsVisible` 方法，那么，如果列表没有重新调整大小以便容纳这个新项，则这个项将是不可见的。因此，通常把对 `ensureIndexIsVisible()` 方法的调用封装在一个 Runnable 中，这个 Runnable 则被传送给 `SwingUtilities.invokeLater()`。

`indexToLocation` 和 `locationToIndex` 方法是把索引转换成位置和把位置转换成索引的两个方便的方法。要了解 `JList.locationToIndex` 方法的使用说明，请参见 17.3.1 节。

(7) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()
public ListUI getUI ()
public String getUIClassID ()
public void setUI (ListUI)
public void updateUI ()
```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

Swing 提示

JList.setPrototypeCellValue () 控制列表的宽度和字体

缺省情况下, 把列表的宽度调整为足以容纳列表中显示的最宽项。对含有许多项的列表而言, 计算列表的宽度会使性能降低。

可以使用 JList.setPrototypeCellValue (Object) 方法把单元宽度和高度显式地设置为对象的首选宽度和高度, 这个对象就是传送给这个方法的参数。对有许多项的列表而言, 当向列表中添加项或从列表中删除项时, 使用 setPrototypeCellValue 方法可以提高性能。另外, 列表的字体也被设置为属性单元值对象所使用的字体。

JList 类还提供了 setFixedCellWidth (int) 和 setFixedCellHeight (int) 方法, 它们把列表单元的宽度和高度设置为传送给这些方法的参数值。

17.3.4 AWT 兼容

Swing 列表和 AWT 列表是两个功能相似的不同组件。虽然这两个组件都显示列表中的项, 但是 JList 比 AWT 的 List 更强壮。AWT 列表只能显示字符串, 而 Swing 列表可以在一个定制单元绘制器中绘制任何内容。AWT 列表只能选取一组相邻的字符串, 而 Swing 列表支持多间隔选取。

表 17-4 列出了由 java.awt.list 实现的 public 方法和与之对应的 JList 方法。

表 17-4 Public java.awt.list 的方法和 JList 的对应方法

java.awt.list 的方法	JList 的对应方法
void add (String)	void DefaultListModel.addElement (Object)
void add (String, int)	void DefaultListModel.add (int, Object)
void addActionListener (ActionListener)	——
void addItemListener (ItemListener)	void addListSelectionListener (ListSelectionListener)
String getItem (int)	list.getModel ().getItemAt (int)
int getItemCount ()	list.getModel ().getSize ()
String [] getItems ()	Object [] DefaultList.getModel ().toArray ()
Dimension getMinimumSize (int)	——
Dimension getPreferredSize (int)	——
int getRows	int getVisibleRowCount ()
int getSelectedIndex ()	int getSelectedIndex ()
int [] getSelectedIndices ()	int getSelectedIndices ()
String getSelectedItem ()	Object getSelectedItem ()
String [] getSelectedItems ()	Object [] getSelectedItem ()
Object [] getSelectedObjects ()	Object [] getSelectedItem ()
int getVisibleIndex ()	——

续表

java.awt.list 的方法	JList 的对应方法
boolean isIndexSelected (int)	boolean isSelectedIndex ()
boolean isMultipleMode ()	int getSelectionMode ()
void makeVisible (int)	void ensureIndexIsVisible (int)
void remove (int)	void DefaultListModel.removeElementAt (int)
void remove (String)	void DefaultListModel.removeElementAt (Object)
void removeActionListener (ActionListener)	——
void removeall ()	void set List Data (null)
void remove ItemListener (ItemListener)	——
void replaceItem (String, int)	void DefaultListModel.removeElementAt (int)
	void DefaultListModel.removeElementAt (Object)
void select (int)	void setSelectedIndex ()
void setMultipleMode (boolean)	void setSelectionMode (int)

当选取项或取消选取时，AWT 激发项事件，当双击一个项时，AWT 激发动作事件。当项被选取或取消选取时，Swing 列表激发列表选取事件，当列表数据被修改时，Swing 列表激发列表数据事件。

JList 没有提供添加项或删除项的方法；因此，等价于 List.add 和 List.remove 方法的 JList 方法与 DefaultListModel 的方法有关。

17.4 本章回顾

本章标志着我们开始接触到 Swing 的较复杂组件。JList 组件是本书介绍的第一个有绘制器的组件。下面三章介绍组合框、表格和树，它们都有绘制器和编辑器。

第 18 章 组 合 框

由 JComboBox 类实现的组合框是由一个可编辑区（缺省时是一个文本区）和一个可选取项的下拉列表组成的。因此，我们采取把它与 JList 比较的方式来介绍 JComboBox。

18.1 JComboBox 与 JList 的比较

JList 和 JComboBox 很相似，因为这两个组件都显示一个项列表。因此，它们都有扩展 ListModel 接口的模型。而且，这两个组件都有绘制器，这些绘制器通过实现 ListCellRenderer 接口来绘制列表单元。

但是，列表和组合框在许多方面还是有差别的。列表单元是不可编辑的，但是组合框可以配备一个编辑器。JComboBox 组件把编辑工作交给实现 ComboBoxEdit 接口的一个对象来处理。

列表支持三个选取模式，并把选取工作交给实现 ListSelectionModel 接口的一个对象来处理。组合框在一个时刻只有一个可选取的项，而且选取工作由组合框模型来处理。另一方面，组合框支持键选取，即在某项上按下一个键就可以选取这个项，但列表不能这样做。

表 18-1 列出了列表和组合框代表的数据类型。

表 18-1 列表和组合框代表的数据类型

代表	JList	JComboBoxEditor
编辑器	——	ComboBoxEditor
模型	ListModel	ComboBoxModel ^①
绘制器	ListCellRenderer	ListCellRenderer
选取模型	DefaultListSelectionModel	——
键选取管理器	——	JComboBox. DefaultKeySelectionManager

① ComboBoxModel 接口扩展 ListModel 接口。

JList 类没有提供任何添加、插入或删除项的方法。在完成构造列表后，唯一修改列表数据的方法是使用 JList.setListData 方法，这个方法允许一次指定所有的项。JComboBox 类正相反，它可以把项添加到组合框中或从组合框中删除项，但是，重新设置组合框中所有项的唯一办法是用 JComboBox.setModel (ComboBoxModel) 来设置它的模型。

18.2 JComboBox 组件

缺省时，JComboBox 的实例是不可编辑的，但是，只需调用 JComboBox.setEditable (true) 就允许进行编辑工作。

图 18-1 示出的小应用程序包含一个组合框和一个控制组合框编辑状态的复选框。

例 18-1 列出了图 18-1 所示的小应用程序的代码。

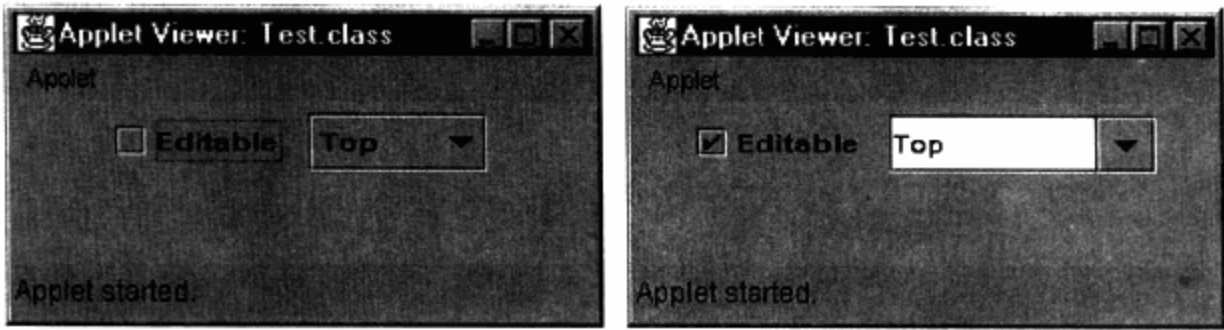


图 18-1 可编辑组合框和不可编辑组合框

例 18-1 可编辑组合框和不可编辑组合框

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JCheckBox checkBox = new JCheckBox ("Editable");
    private JComboBox comboBox = new JComboBox ();

    public void init () {
        Container contentPane = getContentPane ();

        comboBox.addItem ("Top");
        comboBox.addItem ("Center");
        comboBox.addItem ("Bottom");

        checkBox.setSelected (comboBox.isEditable ());

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (checkBox);
        contentPane.add (comboBox);

        checkBox.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                comboBox.setEditable (checkBox.isSelected ());
            }
        });
    }
}
```

这个小应用程序用 `JComboBox` 的无参数构造方法来创建一个组合框。在这个组合框中添加了三个项，然后在这个小应用程序的内容窗格中添加了组合框和复选框。

用从 `JComboBox.setEditable()` 返回的 `boolean` 值来设置这个复选框的初始状态。添加到这个复选框中的动作监听器用从 `JComboBox.isSelected()` 返回的 `boolean` 值来设置组合框的可编辑状态。

18.3 组合框模型

与 `JList` 类一样，`JComboBox` 不维护对它所包含的对象的引用。`JComboBox` 的所有实例都把它们的数据管理工作交给实现 `ComboBoxModel` 接口的一个对象来处理。

可以用下面的 `JComboBox` 构造方法指定在组合框中显示的对象：

- `public JComboBox (ComboBoxModel)`
- `public JComboBox (Object [])`

- public JComboBox (Vector)

还可以在构造后用下面的 JComboBox 方法指定在组合框中显示的对象：

- public void setModel (ComboBoxModel)

图 18-2 示出了组合框模型的类型图。

当把 JComboBox 的实例实例化而且没有显式地指定模型时，这些实例就配备一个模型，这个模型是 DefaultComboBoxModel 的一个实例。DefaultComboBoxModel 类扩展 AbstractListModel 并实现 MutableComboBoxModel 接口。

ListModel（它被 AbstractListModel 和 MutableComboBoxModel 扩展）定义了四个方法，其中两个方法用于添加和删除列表数据监听器，另外两个方法用来获得列表中的元素数和对给定索引的元素的引用。有关 ListModel 接口的更多信息，请参见 17.1 节“列表模型”。

ComboBoxModel 接口扩展 ListModel 并定义两个用于获得和设置当前所选项的方法。

因为组合框总是显示单个所选项，所以，组合框模型必须提供访问这个所选项的方法。

MutableComboBoxModel 是一个接口，它扩展 ComboBoxModel 接口。MutableComboBoxModel 定义了用于添加、插入和删除项的方法。

DefaultComboBoxModel 与 DefaultListModel 有许多共同点。这两个类都扩展 AbstractListModel，而且这两个类都提供数据存储方法，这些数据存储方法是委托给一个矢量实现的。DefaultComboBoxModel 是那些没有显式地指定模型的组合框的缺省模型。

表 18-2 概述了上面介绍的类和接口的功能，并建议在适当的时候使用、实现或扩展它们。

表 18-2 JComboBox 模型

模型	类/接口	功能	当 ... 使用/实现/扩展
ComboBoxModel	接口	所选项的访问方法； 扩展 ListModel	数据是静态的或 MutableComboBoxModel 方法不能满足需要时
MutableComboBoxModel	接口	添加、插入和删除对象	不需要 DefaultComboBoxModel
DefaultComboBoxModel	类	实现 MutableComboBoxModel 并用一个矢量来存储数据	当数据可能修改时（缺省时）

如果组合框数据是可变的，并且可以存储在一个对象数组或一个矢量中，则可以使用缺省的组合框模型，即 DefaultComboBoxModel 的一个实例。

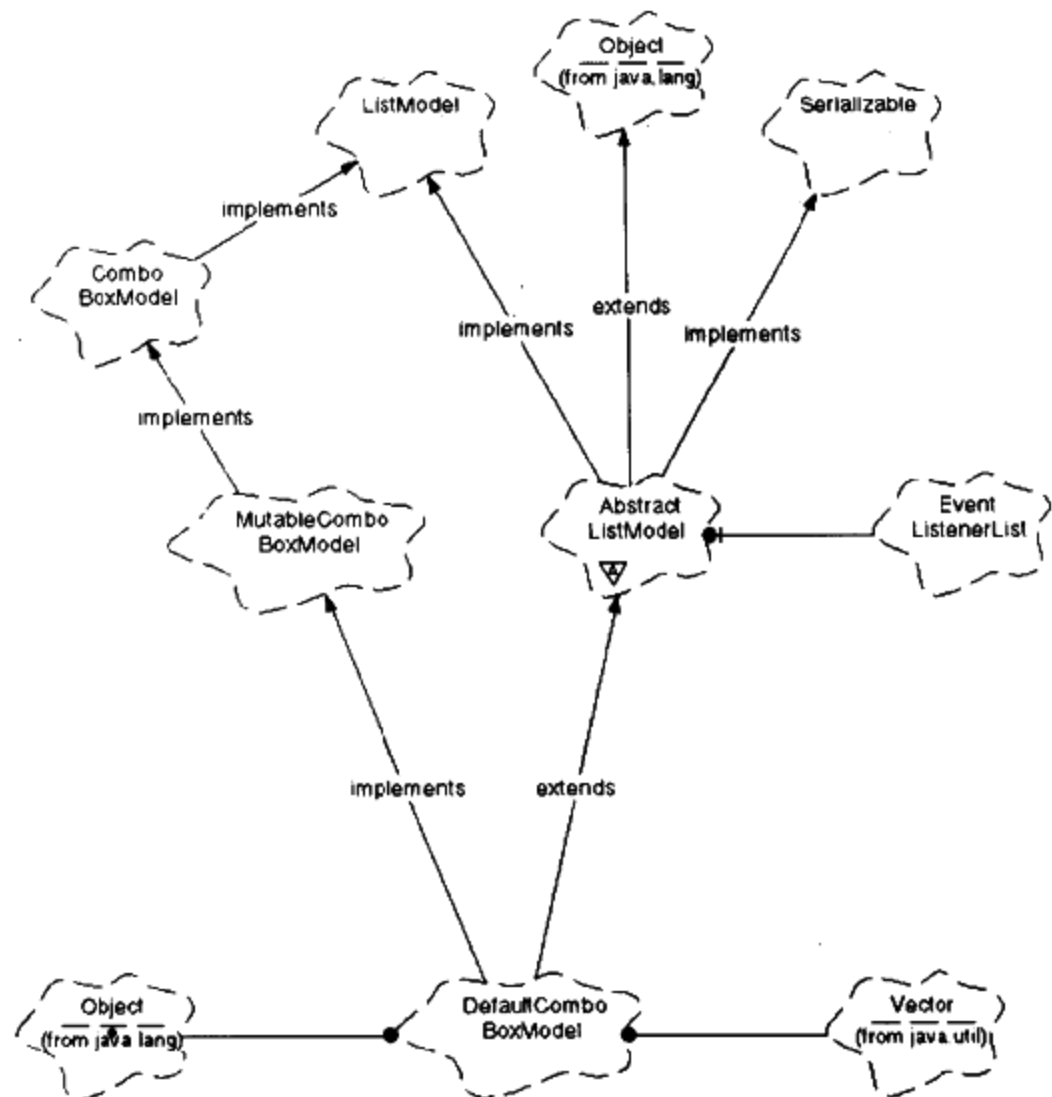


图 18-2 组合框模型

如果数据是可变的,但不能存储到一个对象数组或一个矢量中,则应该使用扩展 `MutableComboBoxModel` 的定制模型。如果数据是静态的,或者如果可变的数据不能由 `MutableComboBoxModel` 接口定义的方法使用,则使用直接扩展 `ComboBoxModel` 的定制组合框模型。

18.3.1 ComboBoxModel

接口总结 18-1 总结了 `ComboBoxModel` 接口。

接口总结 18-1 ComboBoxModel

扩展: `ListModel`

```
public abstract Object getSelectedItem ()
public abstract void setSelectedItem (Object)
```

与列表不同,组合框总是显示单个所选项。因此,必须扩展 `ListModel` 接口,以便把获取和设置当前所选项的方法包括进来。

18.3.2 MutableComboBoxModel

接口总结 18-2 总结了 `MutableComboBoxModel` 接口。

接口总结 18-2 MutableComboBoxModel

```
public abstract void addElement (Object)
public abstract void insertElementAt (Object element, int index)
public abstract void removeElement (Object)
public abstract void removeElementAt (int index)
```

缺省情况下, `ListModel` 和 `ComboBoxModel` 接口都不定义添加、插入或删除列表元素的方法。这组方法由 `MutableComboBoxModel` 接口来定义。

当在组合框中显示的元素可以动态地变化,并且由 `MutableComboBoxModel` 定义的用于操纵数据的这些方法够用时,定制组合框模型将实现 `MutableComboBoxModel` 接口。`DefaultComboBoxModel` 类就是这样的一个模型。

18.3.3 DefaultComboBoxModel

类总结 18-1 总结了 `DefaultComboBoxModel` 类。

类总结 18-1 DefaultComboBoxModel

扩展: `AbstractListModel`

实现: `MutableComboBoxModel`, `java.io.Serializable`

1. 构造方法

```
public DefaultComboBoxModel ()
public DefaultComboBoxModel (Object [])
public DefaultComboBoxModel (Vector)
```

如果用一个对象数组来构造 `DefaultComboBoxModel` 的一个实例,则要分配一个矢量并把对每个对象的引用拷贝到这个矢量中。

如果用无参数的构造方法构造了 `DefaultComboBoxModel` 的实例,则在用下面介绍的 `addElement` 或 `insertElementAt` 方法构造了数据后,应该把这些数据添加到这个模型中。

2. 方法

(1) 元素/大小

```
public Object getElementAt (int)
public int getSize ()

public void addElement (Object)
public void insertElementAt (Object element, int index)
public void removeElement (Object)
public void removeElementAt (int index)
public void removeAllElements ()
```

上面所列的第一组方法由 ListModel 接口定义, ListModel 接口被 ComboBoxModel 接口扩展。DefaultComboBoxModel 是把这些方法委托给一个附属 java.util.Vector 实例来实现它们的。

上面所列的第二组方法由 MutableComboBoxModel 接口定义。与 getSize 和 getElementAt 方法一样, 这些方法是委托给一个附属矢量来实现的。

(2) 所选项/索引

```
public Object getSelectedItem ()
public void setSelectedItem (Object)

public int getIndexOf (Object)
```

上面所列的这些方法由 ComboBoxModel 定义。这些方法提供对组合框所选项的访问。

上面所列的 getIndexOf 和 removeAllElements 都是方便的方法, 任何由 DefaultComboBoxModel 实现的接口都无法将它们定义。这两个方法都是委托给模型的附属矢量来实现的。

18.4 组合框单元绘制器

与列表一样, 组合框用一个列表单元绘制器来绘制单元。事实上, 组合框显示的下拉列表是 JList 的一个实例, 这个实例显示在一个弹出式菜单中。当把列表单元绘制器指定为 JComboBox 的实例时, 组合框用 JList.setCellRenderer() 方法把这个绘制器传递给在弹出式菜单中的列表。

图 18-3 示出的小应用程序包含一个组合框, 这个组合框有一个定制单元绘制器, 它绘制代表一种颜色的文本或图标对。

缺省情况下, 组合框绘制器是 BasicComboBoxRenderer 实例, BasicComboBoxRenderer 在 swing.plaf.basic 包中。BasicComboBoxRenderer 扩展 JLabel 并以这样两种办法中的一种来处理模型值: 用 setIcon 方法来设置绘制器 (它是一个标签) 的图标; 或用 toString () 方法把所有其他类型的对象变成显示在这个标签上的文本。列表和组合框的缺省绘制器以相同的方式来处理数据, 参见“列表单元绘制器”。

虽然 JLabel 能够显示文本和图标, 但是, 因为绘制器的 getListCellRendererComponent 方法只能带一个值, 所以 BasicComboBoxRenderer 只能显示文本或者图标。带入了的值如果是字符串或图标, 则可以按原样绘制, 但所有其他类型的对象 (例如, 包括一个带字符串和图标的 Object 数组) 就必须用从这些对象的 toString 方法返回的字符串来绘制。因此, 要同时绘制文本和图像, 就必须实现一个定制绘制器。

图 18-3 示出的小应用程序用 JComboBox (Object []) 构造方法创建一个组合框。数组中的

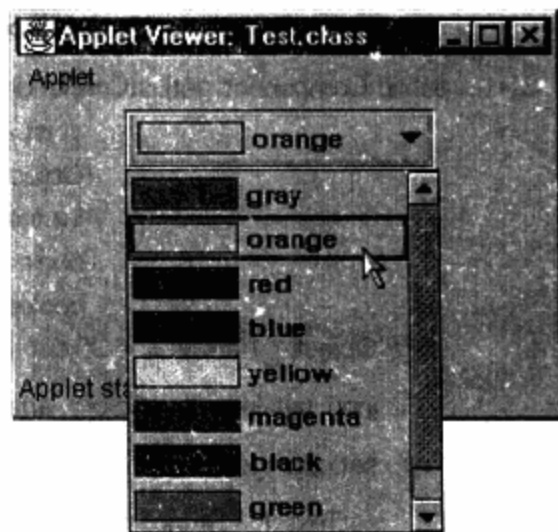


图 18-3 带一个定制列表单元绘制器的组合框

每一项都是一个 Object 数组，这个 Object 数组包含一个颜色和一个字符串。

```
public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JComboBox combo = new JComboBox (new Object [] {
            new Object [] { Color.gray, "gray" },
            new Object [] { Color.orange, "orange" },
            new Object [] { Color.red, "red" },
            new Object [] { Color.blue, "blue" },
            new Object [] { Color.yellow, "yellow" },
            new Object [] { Color.magenta, "magenta" },
            new Object [] { Color.black, "black" },
            new Object [] { Color.green, "green" },
            new Object [] { Color.lightGray, "lightGray" } });
        combo.setRenderer (new ColorRenderer ());
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (combo);
    }
}
```

把这个组合框的绘制器设置为一个 ColorRenderer 实例，这个实例绘制文本和图标。

```
class ColorRenderer extends JLabel implements ListCellRenderer {
    private static ColorIcon icon = new ColorIcon ();
    ...
}
```

ColorRenderer 类扩展 JLabel 并实现 ListCellRenderer 接口。ColorRenderer 包含一个 static ColorIcon 实例，这个实例用于在列表单元绘制有颜色的矩形。ColorIcon 是 Icon 接口的一个简单实现。因为组件可以共享图标，所以，可以用单个图标来绘制列表单元中有颜色的矩形。有关在组件中共享图标的更多信息，请参见 5.2.2 节“在组件中共享图标”。

```
...
private Border
    redBorder = BorderFactory.createLineBorder (Color.red, 2),
    emptyBorder = BorderFactory.createEmptyBorder (2, 2, 2, 2);

public Component getListCellRendererComponent (
    JList list,
    Object value,
    int index,
    boolean isSelected,
    boolean cellHasFocus) {
    Object [] array = (Object []) value;
    icon.setColor ((Color) array [0]);
    setIcon (icon);
    setText ((String) array [1]);
    if (isSelected) setBorder (redBorder);
    else setBorder (emptyBorder);
    return this;
}
```

这个绘制器创建了两个边框，它们指出一个单元是否可选取。如果选取了正在绘制的单元，则用红边框来指示选取，否则将使用空边框。

传送给 `getListCellRendererComponent()` 的值是存储在组合框模型中的一个值。在这种情况下，这个值是在创建组合框时指定的一个 `Object` 数组，这个绘制器从这个数组中提取颜色并用这个颜色来设置图标的颜色。然后，把图标指定为这个标签的图标（回忆一下，`ColorRenderer` 扩展 `JLabel`），并把存储在这个数组中的字符串设置为这个标签的文本。

例 18-2 完整地列出了图 18-3 所示小应用程序的代码。

例 18-2 一个定制列表单元绘制器

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JComboBox combo = new JComboBox (new Object [] {
            new Object [] { Color.gray, "gray" },
            new Object [] { Color.orange, "orange" },
            new Object [] { Color.red, "red" },
            new Object [] { Color.blue, "blue" },
            new Object [] { Color.yellow, "yellow" },
            new Object [] { Color.magenta, "magenta" },
            new Object [] { Color.black, "black" },
            new Object [] { Color.green, "green" },
            new Object [] { Color.lightGray, "lightGray" } });

        combo.setRenderer (new ColorRenderer ());

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (combo);
    }
}

class ColorRenderer extends JLabel implements ListCellRenderer {
    private static ColorIcon icon = new ColorIcon ();

    private Border
        redBorder = BorderFactory.createLineBorder (Color.red, 2),
        emptyBorder = BorderFactory.createEmptyBorder (2, 2, 2, 2);

    public Component getListCellRendererComponent (
        JList list,
        Object value,
        int index,
        boolean isSelected,
        boolean cellHasFocus) {

        Object [] array = (Object []) value;
        icon.setColor ( (Color) array [0]);

        setIcon (icon);
        setText ( (String) array [1]);

        if (isSelected) setBorder (redBorder);
        else setBorder (emptyBorder);

        return this;
    }
}
```

```

|
class ColorIcon implements Icon {
    private Color color;
    private int w, h;

    public ColorIcon () {
        this (Color.gray, 50, 15);
    }

    public ColorIcon (Color color, int w, int h) {
        this.color = color;
        this.w = w;
        this.h = h;
    }

    public void paintIcon (Component c, Graphics g, int x, int y) {
        g.setColor (Color.black);
        g.drawRect (x, y, w-1, h-1);
        g.setColor (color);
        g.fillRect (x+1, y+1, w-2, h-2);
    }

    public Color getColor () {
        return color;
    }

    public void setColor (Color color) {
        this.color = color;
    }

    public int getIconWidth () {
        return w;
    }

    public int getIconHeight () {
        return h;
    }
}
|

```

18.5 组合框键选取管理器

组合框允许用按下一个键来选取项。如果在组合框有焦点时按下了一个键，则开始搜索这个键与组合框列表中项的匹配，如果找到一个匹配，则匹配的项就可选取。JComboBox 把搜索和匹配的工作交给一个实现 JComboBox.KeySelectionManager 接口的对象来完成。接口总结 18-3 中列出了 JComboBox.KeySelectionManager 接口。

接口总结 18-3 JComboBox.KeySelectionManager

```
public abstract int selectionForKey (char key, ComboBoxModel model)
```

JComboBox 的每个实例都维护对一个对象的引用，这个对象实现 KeySelectionManager 接口。当一个键被按下时，就调用键选取管理器的 selectionForKey 方法，以便获得选取项的索引。如果 selectionForKey 返回一个大于-1的索引，则可选取相应的项。

缺省情况下，组合框的键选取管理器是 JComboBox.DefaultKeySelectionManager 的一个实例。JComboBox.DefaultKeySelectionManager 以两种路径来搜索项，如图 18-4 所示。第一种路径从当前

- 如果所选项是列表的最后项，则搜索从第一项开始。

所选项下面的一项开始，搜索到列表的最后一项[○]。如果按下的键与所有的项都不匹配，则使用第二种搜索路径，它从第一项开始，搜索到所选项上面的一项。

例如，对图 18-4 示出的组合框列表来说，在选取 yellow 时，按下了 b 键，将导致选取 black。接下来按下 b 键将选取 blue。blue 选取是最值得注意的，因为它需要两种搜索路径。

JComboBox.DefaultKeySelectionManager 类把一个匹配定义为一个键与第一个字符（这个字符是从一个项的 toString 方法中返回的字符串的第一个字符）之间的一个匹配。

KeySelectionManager 接口和 DefaultKeySelectionManager 类是嵌套在 JComboBox 类中的，因为它们只有在组合框中才有意义（人们可能会为列表争取同样的功能，列表不提供键选取）。

JComboBox.DefaultKeySelectionManager 类具有包范围，即它只可以由 swing 包中的其他类访问。JComboBox.KeySelectionManager 接口是 public，以便它可以在 swing 包外实现。

18.5.1 使用缺省键选取管理器

图 18-4 给出了图 18-3 中示出的小应用程序的组合框列表。然而，图 18-3 所示的组合框没有对键按下作出反应，因为包含在组合框模型中的项是 Object 数组。

缺省键选取管理器把一个键与从一个项的 toString 方法中返回的字符串的第一个字符相匹配。从一个 Object 数组的 toString 方法中返回的字符串是标识符，其形式类似于下面的内容：[Ljava.lang.Object; @f5f0e605。因为这个标识符与用项表示的颜色没有关系，所以，图 18-3 所示的小应用程序的键选取没有起作用[○]。

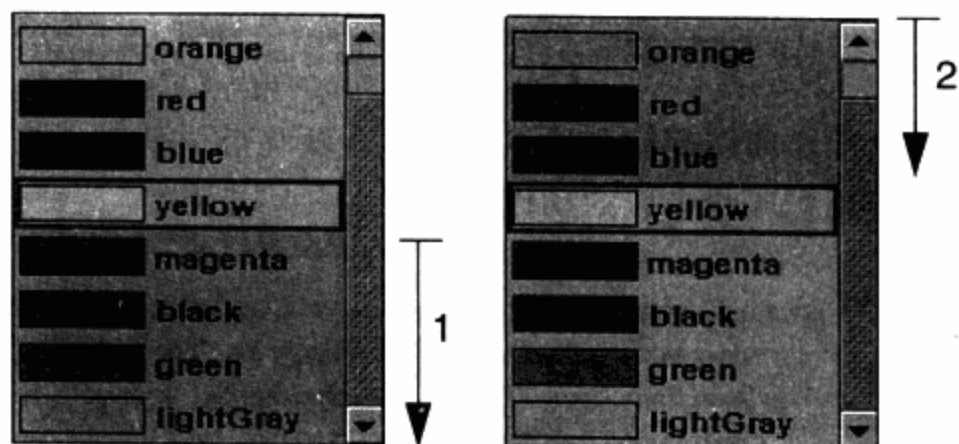


图 18-4 缺省键选取管理器的搜索规则

初看上去，似乎用一个定制的关键选取管理器是符合需求的，因为定制键选取器把一个键与项数组所包含的字符串的第一个字符相匹配。然而，一个简单的解决方法是把这个 Object 数组封装在一个实现 toString 方法的类中，下面列出了这样的一个类。

```
Class Item {
    private Color color;
    private String string;

    public Item (Object [] array) {
        color = (Color) array [0];
        string = (String) array [1];
    }

    public Color getColor () { return color; }
    public String toString () { return string; }
}
```

用一个 Object 数组来构造 Item 的实例，这个 Object 数组包含一个颜色和一个字符串（代表颜色）。从这个 Object 数组中提取颜色和字符串，并分别通过 getColor 和 toString 方法来获得这个颜色和这个字符串。

[○] 事实上，因为所有的标识符都以相同的字符开始，所以按下键不能移动选取

Item 类最重要的方面是它的 toString 方法。虽然由于上述原因，组合框的缺省键选取管理器不能与 Object 数组很好地一起工作，但是，缺省键选取管理器确实能与实现一个相应的 toString 方法（如由 Item 类实现的 toString 方法）的类一起很好的工作。

图 18-5 示出的小应用程序包含一个组合框，它的项是 Item 类的实例。当组合框有焦点时按下了一个键，则与相应的项相关联的键和字符串会显示在小应用程序的状态区中。

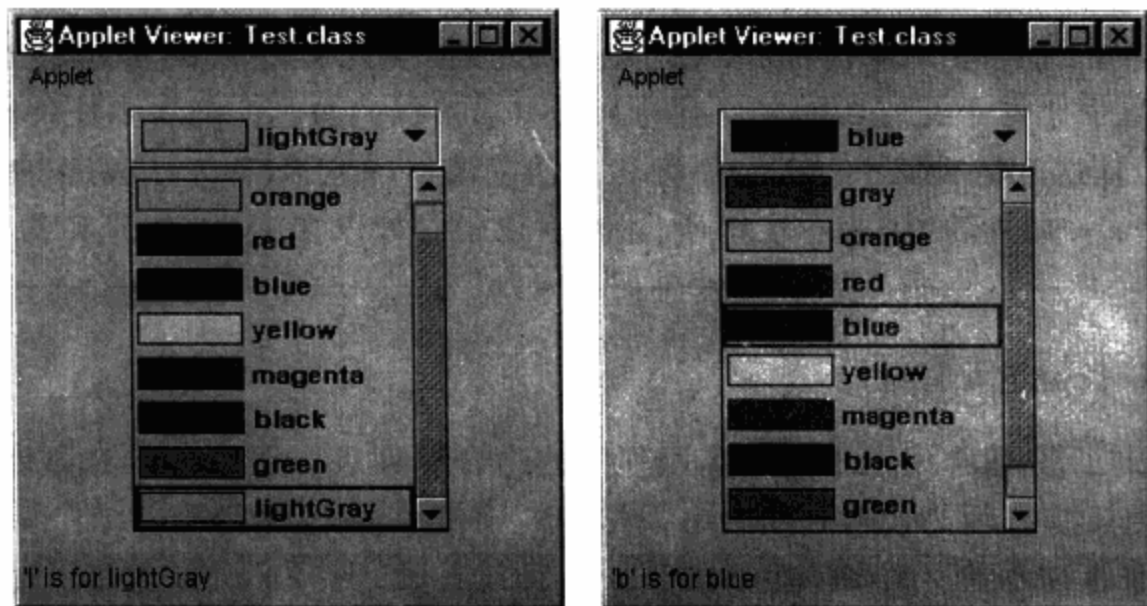


图 18-5 使用组合框的缺省键选取管理器

这个小应用程序创建了一个组合框（带一个 Item 实例数组），并在这个组合框中添加了更新小应用程序状态区的动作监听器。

```
public class Test extends JApplet {
    private JComboBox colorCombo = new JComboBox (new Object [] {
        new Item (new Object [] { Color.gray, "gray" }),
        new Item (new Object [] { Color.orange, "orange" }),
        new Item (new Object [] { Color.red, "red" }),
        new Item (new Object [] { Color.blue, "blue" }),
        new Item (new Object [] { Color.yellow, "yellow" }),
        new Item (new Object [] { Color.magenta, "magenta" }),
        new Item (new Object [] { Color.black, "black" }),
        new Item (new Object [] { Color.green, "green" }),
        new Item (new Object [] { Color.lightGray, "lightGray" })
    });

    public void init () {
        final Container contentPane = getContentPane ();
        colorCombo.setRenderer (new ColorRenderer ());
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (colorCombo);

        colorCombo.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                Item item = (Item) colorCombo.getSelectedItem ();
                String first = item.toString ().substring (0, 1);
                showStatus (" " + first + " " + " is for " + item);
            }
        });
    }
}
```


这个监听器通过调用 `JComboBox.getSelectedItem()` 来获得对所选项的一个引用, 而且通过调用 `Item.toString()`, 然后调用 `String.substring(0, 1)` 来获得这个项的字符串的第一个字符。

例 18-3 完整地列出了图 18-5 所示小应用程序的代码。

例 18-3 使用缺省的键选取管理器

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet {
    private JComboBox colorCombo = new JComboBox (new Object [] {
        new Item (new Object [] { Color.gray, "gray" }),
        new Item (new Object [] { Color.orange, "orange" }),
        new Item (new Object [] { Color.red, "red" }),
        new Item (new Object [] { Color.blue, "blue" }),
        new Item (new Object [] { Color.yellow, "yellow" }),
        new Item (new Object [] { Color.magenta, "magenta" }),
        new Item (new Object [] { Color.black, "black" }),
        new Item (new Object [] { Color.green, "green" }),
        new Item (new Object [] { Color.lightGray, "lightGray" })
    });

    public void init () {
        final Container contentPane = getContentPane ();
        colorCombo.setRenderer (new ColorRenderer ());
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (colorCombo);
        colorCombo.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                Item item = (Item) colorCombo.getSelectedItem ();
                String first = item.toString ().substring (0, 1);
                showStatus (" " + first + " " + " is for " + item);
            }
        });
    }

    class Item {
        private Color color;
        private String string;

        public Item (Object [] array) {
            color = (Color) array [0];
            string = (String) array [1];
        }

        public Color getColor () { return color; }
        public String toString () { return string; }
    }
}
```

18.5.2 定制键选取管理器

当由 `JComboBox.DefaultKeySelectionManager` 类实现的键/项匹配算法不能满足需要时, 则可

以实现 `JComboBox.KeySelectionManager` 接口以便提供一个定制的键选取管理器。

我们知道，把键与项匹配的缺省算法就是将每一个键与从项的 `toString` 方法返回的字符串的第一个字符进行比较。

例如，如果在图 18-6 所示的组合框中选取了最上面的项，那么输入 `blu` 导致下面的选取：`b` 选取 `blue` (蓝色)，`l` 选取 `light gray`，`u` 什么也不选取，因为没有以 `u` 开头的颜色。

图 18-6 示出的小应用程序包含一个组合框，这个组合框有一个定制的键选取管理器，它用一个字符串来存储时间间隔在半

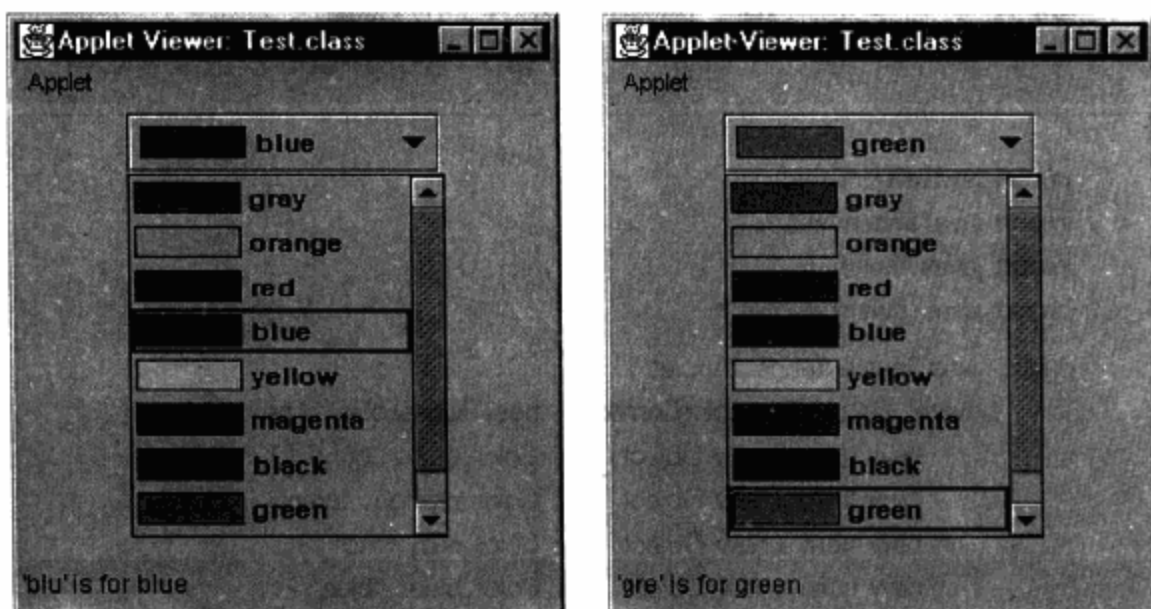


图 18-6 实现一个定制的键选取管理器

秒钟之内连续的键击输入，用这个字符串而不是单个键击来进行比较。例如，连续输入 `blu` (每次键击的时间间隔不大于半秒) 将选取 `blue`，不选取 `light gray`，这与缺省键选取管理器 (`JComboBox.DefaultKeySelectionManager`) 选取的情况不同。

这个小应用程序创建一个组合框 (带一个 `Item` 实例数组)。为方便起见，这个 `Item` 数组可重复使用。

```
class Item {
    private Color color;
    private String string;

    public Item (Object [] array) {
        color = (Color) array [0];
        string = (String) array [1];
    }

    public Color getColor () { return color; }
    public String toString () { return string; }
}
```

这个小应用程序把组合框的键选取管理器设置为 `ColorKeySelectionManager` 的一个实例 (如下所列)，而且把一个动作监听器添加到这个组合框中。当选取了组合框中的一个项时，这个监听器将更新这个小应用程序的状态条。

```
public class Test extends JApplet {
    private ColorKeySelectionManager ksm =
        new ColorKeySelectionManager ();

    private JComboBox colorCombo = new JComboBox (new Object [] {
        new Item (new Object [] { Color.gray, "gray" }),
        new Item (new Object [] { Color.orange, "orange" }),
        new Item (new Object [] { Color.red, "red" }),
        new Item (new Object [] { Color.blue, "blue" }),
        new Item (new Object [] { Color.yellow, "yellow" }),
        new Item (new Object [] { Color.magenta, "magenta" }),
        new Item (new Object [] { Color.black, "black" }),
        new Item (new Object [] { Color.green, "green" }),
    });
}
```

```

        new Item (new Object [] { Color.lightGray, "lightGray"})
    );
    public void init () {
        final Container contentPane = getContentPane ();
        colorCombo.setRenderer (new ColorRenderer ());
        colorCombo.setKeySelectionManager (ksm);
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (colorCombo);

        colorCombo.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                Item item = (Item) colorCombo.getSelectedItem ();
                String itemString = item.toString ();
                showStatus (" " + ksm.getSearchString () +
                    " " + " is for " + itemString);
            }
        });
    }
}

```

ColorKeySelectionManager 类实现 JComboBox.KeySelectionManager 接口并维护一个搜索字符串和一个时间标志。selectionForKey 方法（它在 JComboBox.keySelectionManager () 中定义）更新这个搜索字符串并执行从所选项后面一个项到列表中最后一项的搜索工作。如果这种搜索证明是不成功的而且没有从列表头开始，则执行另一种搜索，它从列表的头开始^①。

```

class ColorKeySelectionManager
    implements JComboBox.KeySelectionManager {
    private String searchString = new String ();
    private long lastTime;

    public int selectionForKey (char key, ComboBoxModel model) {
        updateSearchString (model, key);

        int start = findindex (model, getSelectedString (model));
        int selection = search (model, start);

        if (selection == -1 && start != 0)
            selection = search (model, 0);

        return selection;
    }

    public String getSearchString () {
        return searchString;
    }

    public int search (JComboBoxModel model, int start) {
        return -1;
    }

    private int findIndex (ComboBoxModel model, String find) {
        int size = model.getSize ();

        if (find != null) {
            for (int i=0; i < size; ++i) {
                String s = getString (model, i);
            }
        }
    }
}

```

① 为简单起见，第二种搜索办法搜索整个列表。这种搜索办法应该在搜索到所选项上面的项后就结束。

```

        if (s.compareToIgnoreCase (find) == 0) {
            return (i == size-1) ? i+1;
        }
    }
    return 0;
}

private String getSelectedString (ComboBoxModel model) {
    Item item = (Item) model.getSelectedItem ();
    return item.toString ();
}

private void updateSearchString (
    ComboBoxModel model, char key) {
    long time = System.currentTimeMillis ();
    if (time - lastTime < 500) searchString += key;
    else searchString = "" + key;
    lastTime = time;
}

```

这里提供了 `getSearchString` 方法，以便更新这个小应用程序状态条的动作监听器能很容易地访问到这个搜索字符串。

由 `ColorKeySelectionManager` 类实现的其他方法是 `selectionForKey` 方法的支持方法，在下面的程序中列出了它们，但没有在这里对它们进行说明。

```

private int search (ComboBoxModel model, int start) {
    for (int i = start; i < model.getSize (); ++i) {
        String s = getString (model, i);
        int searchLength = searchString.length ();
        if (s.regionMatches (0, searchString, 0, searchLength))
            return i;
    }
}

```

例 18-4 完整地列出了图 18-6 所示的小应用程序的代码。

例 18-4 实现一个定制的键选取管理器

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet {
    private ColorKeySelectionManager ksm =
        new ColorKeySelectionManager ();

    private JComboBox colorCombo = new JComboBox (new Object [] {
        new Item (new Object [] { Color.gray, "gray" }),
        new Item (new Object [] { Color.orange, "orange" }),
        new Item (new Object [] { Color.red, "red" }),
        new Item (new Object [] { Color.blue, "blue" }),
        new Item (new Object [] { Color.yellow, "yellow" }),
        new Item (new Object [] { Color.magenta, "magenta" }),
        new Item (new Object [] { Color.black, "black" }),
        new Item (new Object [] { Color.green, "green" }),
        new Item (new Object [] { Color.lightGray, "lightGray" })
    });
}

```

```

public void init () {
    final Container contentPane = getContentPane ();
    colorCombo.setRenderer (new ColorRenderer ());
    colorCombo.setKeySelectionManager (ksm);
    contentPane.setLayout (new FlowLayout ());
    contentPane.add (colorCombo);
    colorCombo.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            Item item = (Item) colorCombo.getSelectedItem ();
            String itemString = item.toString ();
            showStatus (" " + ksm.getSearchString () +
                " " + " is for " + itemString);
        }
    });
}

class Item {
    private Color color;
    private String string;

    public Item (Object [] array) {
        color = (Color) array [0];
        string = (String) array [1];
    }

    public Color getColor () { return color; }
    public String toString () { return string; }
}

class ColorKeySelectionManager
    implements JComboBox.KeySelectionManager {
    private String searchString = new String ();
    private long lastTime;

    public int selectionForKey (char key, ComboBoxModel model) {
        updateSearchString (model, key);

        int start = findIndexAfter (model, getSelectedString (model));
        int selection = search (model, start);

        if (selection == -1 && start != 0)
            selection = search (model, 0);

        return selection;
    }

    public String getSearchString () {
        return searchString;
    }

    private int search (ComboBoxModel model, int start) {
        for (int i = start; i < model.getSize (); ++i) {
            String s = getString (model, i);
            int searchLength = searchString.length ();

            if (s.regionMatches (0, searchString, 0, searchLength))
                return i;
        }

        return -1;
    }
}

```

```
private int findIndex (ComboBoxModel model, String find) {
    int size = model.getSize ();
    if (find != null) {
        for (int i=0; i < size; ++i) {
            String s = getString (model, i);
            if (s.compareToIgnoreCase (find) == 0) {
                return (i == size-1) ? 0 : i + 1;
            }
        }
    }
    return 0;
}

private String getString (ComboBoxModel model, int index) {
    Item item = (Item) model.getElementAt (index);
    return item.toString ();
}

private String getSelectedString (ComboBoxModel model) {
    Item item = (Item) model.getSelectedItem ();
    return item.toString ();
}

private void updateSearchString (
    ComboBoxModel model, char key) {
    long time = System.currentTimeMillis ();
    if (time - lastTime < 500) searchString += key;
    else
        searchString = "" + key;
    lastTime = time;
}
```

18.5.3 程序式的键选取

当一个组合框有焦点时，可以用 JComboBox.selectWithKeyChar (char) 方法来程序地执行按下一个键的操作。selectWithKeyChar 方法以代表一个键的字符为参数，并用组合框的键选取管

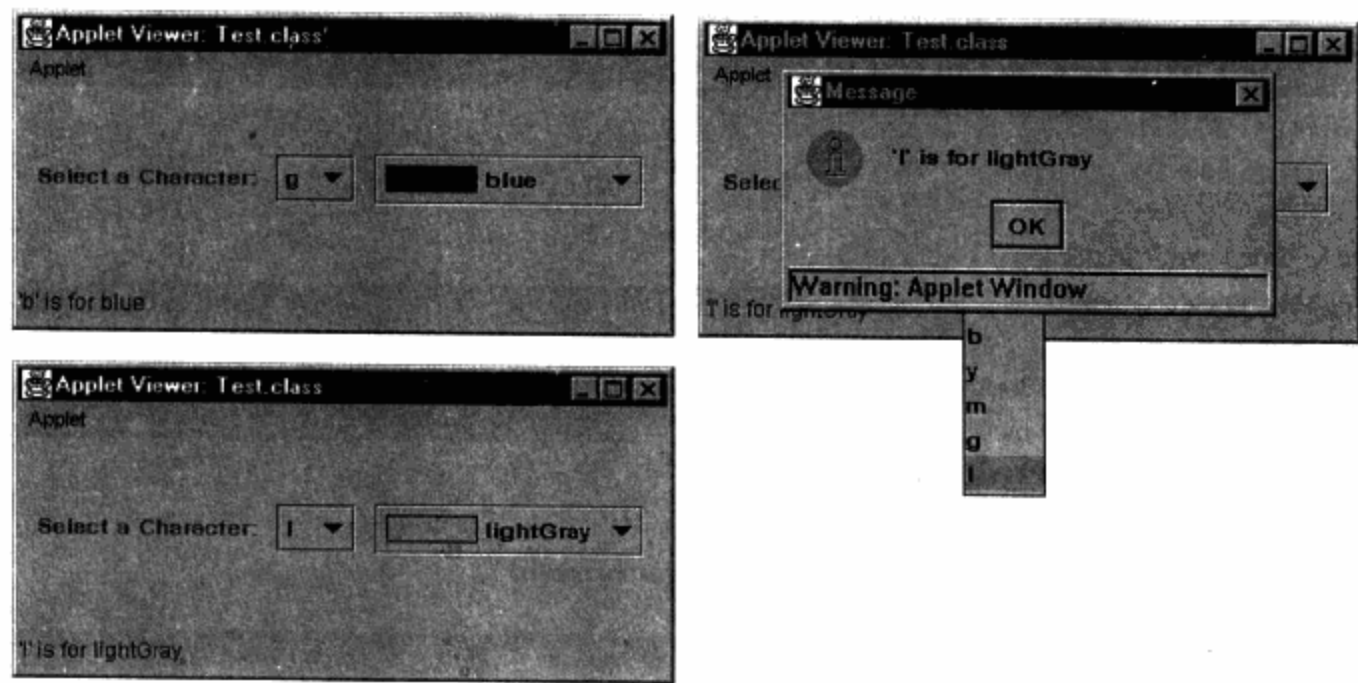


图 18-7 使用一个键来选取组合框中的项

理器来选取一个相应的项。

图 18-7 示出的小应用程序包含两个组合框。通过把所选取的字符传送给 `JComboBox.selectWithChar()` 方法, 用左边组合框选取的字符选取右边组合框的项。当在右边组合框选取一个项时, 则显示一个消息对话框并指出所选的项和用于选取它的字符。

图 18-7 中左边的图片显示这个小应用程序开始的样子。右边的图片显示从左组合框中选取了“l”, 并显示了消息对话框后小应用程序的样子。最下面的图片示出了清除消息对话框后的情况。

这个小应用程序创建了两个组合框。左边的组合框包含 `java.lang.Character` 的实例, 右边的组合框包含 `Item` 的实例。

```
public class Test extends JApplet {
    private JComboBox charsCombo = new JComboBox (new Object [] {
        new Character (' g'), new Character (' o'),
        new Character (' r'), new Character (' b'),
        new Character (' y'), new Character (' m'),
        new Character (' g'), new Character (' l'),
    });
    private JComboBox colorCombo = new JComboBox (new Object [] {
        new Item (new Object [] { Color.gray, "gray" }),
        new Item (new Object [] { Color.orange, "orange" }),
        new Item (new Object [] { Color.red, "red" }),
        new Item (new Object [] { Color.blue, "blue" }),
        new Item (new Object [] { Color.yellow, "yellow" }),
        new Item (new Object [] { Color.magenta, "magenta" }),
        new Item (new Object [] { Color.black, "black" }),
        new Item (new Object [] { Color.green, "green" }),
        new Item (new Object [] { Color.lightGray, "lightGray" })
    });
}
```

这两个组合框中添加了动作监听器。字符组合框通过调用颜色组合框的 `selectWithKeyChar`, 然后显示消息对话框来响应选取。颜色组合框通过更新这个小应用程序的状态区来响应选取。

```
public void init () {
    final Container contentPane = getContentPane ();
    colorCombo.setRenderer (new ColorRenderer ());
    charsCombo.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            final Character c = (Character)
                charsCombo.getSelectedItem ();
            colorCombo.selectWithKeyChar (c.charValue ());
            Item item = (Item) colorCombo.getSelectedItem ();
            JOptionPane.showMessageDialog (contentPane,
                "" + c.toString () + "" +
                " is for " + item.toString ());
        }
    });
    colorCombo.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            Item item = (Item) colorCombo.getSelectedItem ();
        }
    });
}
```



```

        Character first = new Character (
            Item.toString () .charAt (0));
        ShowStatus (" " + first.toString () + " " +
            " is for " + item);
    }

    );

    contentPane.setLayout (
        new FlowLayout (FlowLayout.CENTER, 10, 35));
    contentPane.add (new JLabel ("Select a Character:"));
    contentPane.add (charsCombo);
    contentPane.add (colorCombo);
}

```

从图 18-7 中可以明显地看出, 右边的图片中有错误, 因为字符组合框的下拉列表在对话框下面显示出来。通过把对 `JOptionPane.showMessageDialog()` 的调用封装在一个 `Runnable` 中, 以便在这个对话框显示前擦除那个不好看的下拉列表, 如图 18-8 所示, 其中, 这个 `Runnable` 被传送给 `SwingUtilities.invokeLater()` 方法。

```

charsCombo.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        final Character c = (Character)
            charsCombo.getSelectedItem ();
        colorCombo.selectWithKeyChar (c.charValue ());
        SwingUtilities.invokeLater (new Runnable () {
            public void run () {
                Item item = (Item) colorCombo.getSelectedItem ();
                JOptionPane.showMessageDialog (contentPane,
                    " " + c.toString () + " " +
                    " is for " + item.toString ());
            }
        });
    }
});

```

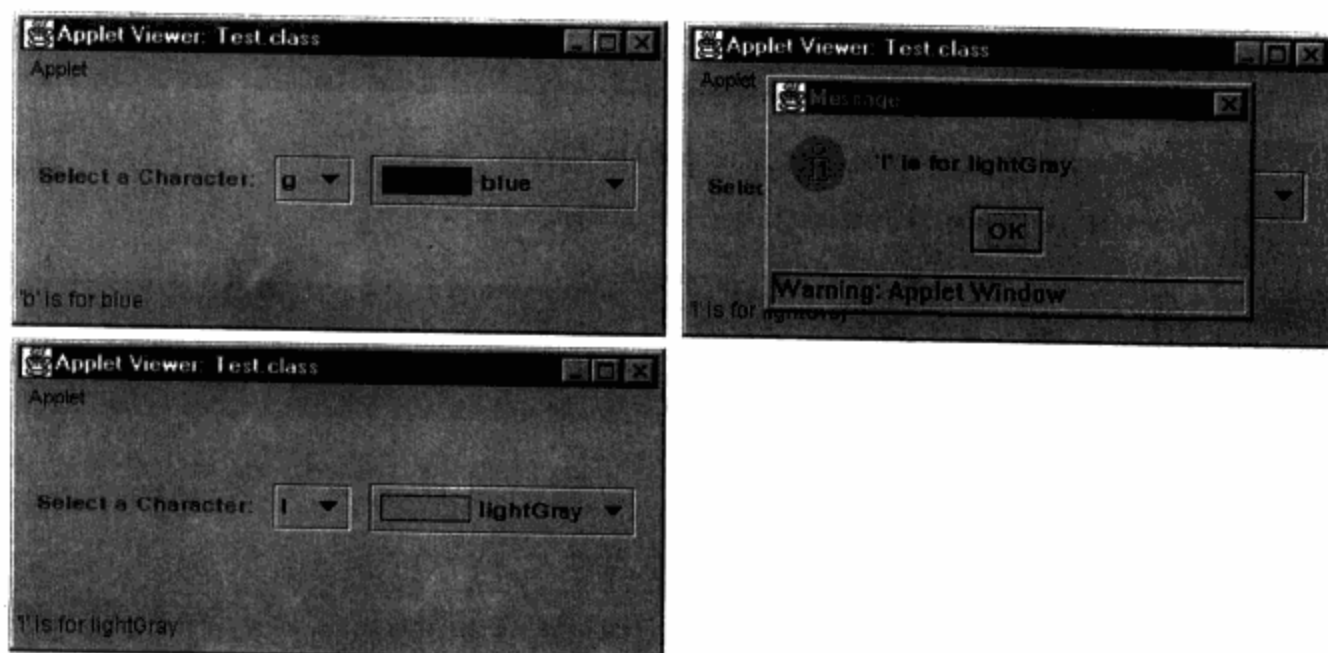


图 18-8 使用 `SwingUtilities.invokeLater()`

如上所列，当从一个 Runnable 中显示这个对话框时，其结果如图 18-8 所示，其中，这个 Runnable 会传送给 SwingUtilities.invokeLater() 方法。

例 18-5 完整地列出了图 18-8 示出的小应用程序的代码。

例 18-5 使用 SwingUtilities.invokeLater ()

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Test extends JApplet {
    private JComboBox charsCombo = new JComboBox (new Object [] {
        new Character (' g'), new Character (' o'),
        new Character (' r'), new Character (' b'),
        new Character (' y'), new Character (' m'),
        new Character (' g'), new Character (' l'),
    });
    private JComboBox colorCombo = new JComboBox (new Object [] {
        new Item (new Object [] { Color.gray, "gray" }),
        new Item (new Object [] { Color.orange, "orange" }),
        new Item (new Object [] { Color.red, "red" }),
        new Item (new Object [] { Color.blue, "blue" }),
        new Item (new Object [] { Color.yellow, "yellow" }),
        new Item (new Object [] { Color.magenta, "magenta" }),
        new Item (new Object [] { Color.black, "black" }),
        new Item (new Object [] { Color.green, "green" }),
        new Item (new Object [] { Color.lightGray, "lightGray" })
    });
    public void init () {
        final Container contentPane = getContentPane ();
        colorCombo.setRenderer (new ColorRenderer ());
        colorCombo.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                Item item = (Item) colorCombo.getSelectedItem ();
                Character first = new Character (
                    item.toString ().charAt (0));
                showStatus (" " + first.toString () + " " +
                    " is for " + item);
            }
        });
        charsCombo.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                final Character c = (Character)
                    charsCombo.getSelectedItem ();
                colorCombo.selectWithKeyChar (c.charValue ());
                Item item = (Item) colorCombo.getSelectedItem ();
                JOptionPane.showMessageDialog (contentPane,
                    " " + c.toString () + " " +
                    " is for " + item.toString ());
            }
        });
    }
}
```

```
contentPane.setLayout (
    new FlowLayout (FlowLayout.CENTER, 10, 35));

contentPane.add (new JLabel ("Select a Character:"));
contentPane.add (charsCombo);
contentPane.add (colorCombo);
}

class Item {
    private Color color;
    private String string;

    public Item (Object [] array) {
        color = (Color) array [0];
        string = (String) array [1];
    }

    public Color getColor () { return color; }
    public String toString () { return string; }
```

18.6 组合框编辑器

与列表不同，组合框是可以编辑的。用 `JComboBox.setEditable` 方法来控制组合框是否可编辑。缺省情况下，`JComboBox` 的实例是不可编辑的，在对组合框进行构造后，为了使这个组合框是可编辑的，必须调用 `setEditable (true)`。`JComboBox.isEditable ()` 返回一个 `boolean` 值来指示一个组合框的可编辑性。

组合框提供了一个可视的提示标记，指出组合框是否可编辑，如图 18-9 所示。图 18-9 中显示的是标准的 Swing 界面样式的组合框，右边图片中的组合框是可编辑的，而左边图片中的组合框是不可编辑的。

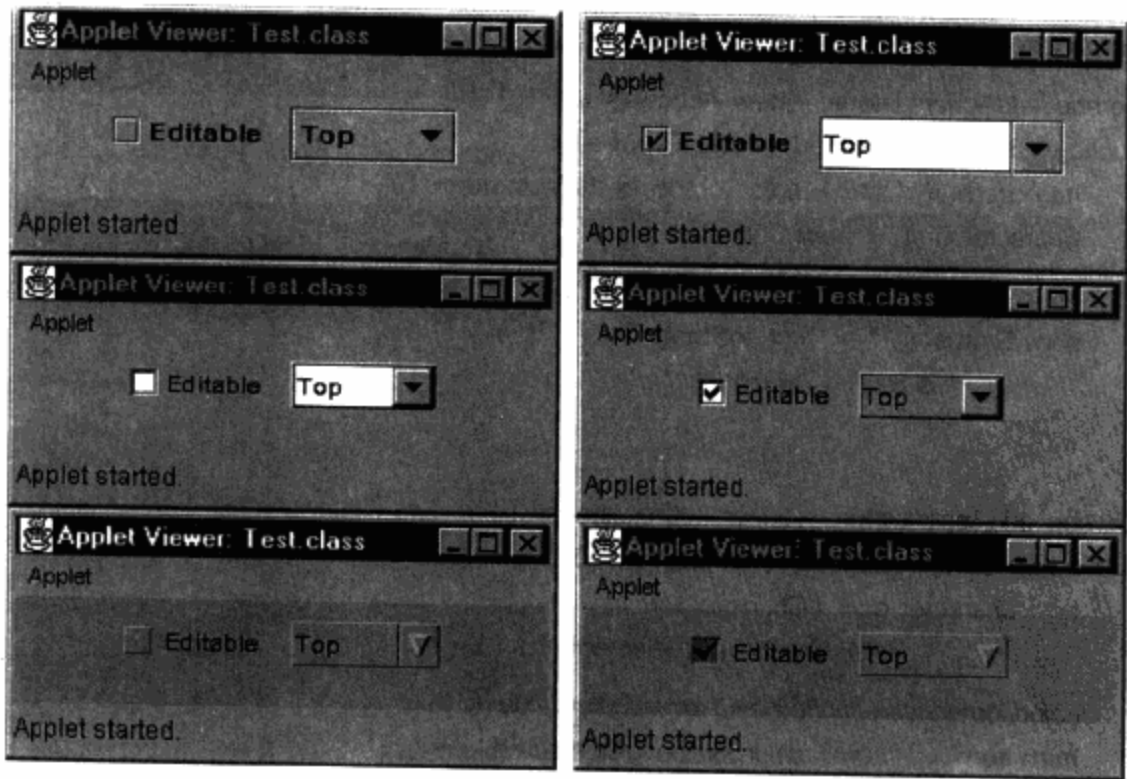


图 18-9 可编辑和不可编辑复选框

可编辑（即调用 `JComboBox.setEditable (true)`）将使一个组件（由编辑器提供的）添加到这个组合框。缺省情况下，由 `BasicComboBoxEditor` 的一个实例来提供一个文本域。

图 18-10 示出了在激活 `JComboBox.setEditable(true)` 时发生的事件序列。`JComboBox` 的可编辑属性是一个关联属性，因此，当修改组合框的可编辑状态时，`JComboBox` 的实例就激发属性变化事件。属性变化最终由这个组合框的 UI 代表来处理，它把编辑器的组件添加到这个组合框中。

没有为组合框精确地定义可编辑的确切含义。缺省的组合框编辑器（`BasicComboBoxEditor` 的一个实例）编辑已选取的项，而组合框列表中显示的项不变化。

例如，图 18-11 示出了一个正在编辑的组合框。左上角的图片显示处于初始状态的组合框。在右上角的图片中，选取项已经编辑，但是这次编辑还没有由用户按下回车键来提交。左下角的图片是按下回车键后的样子，右下角的图片是接着在组合框中进行了选取后的样子。

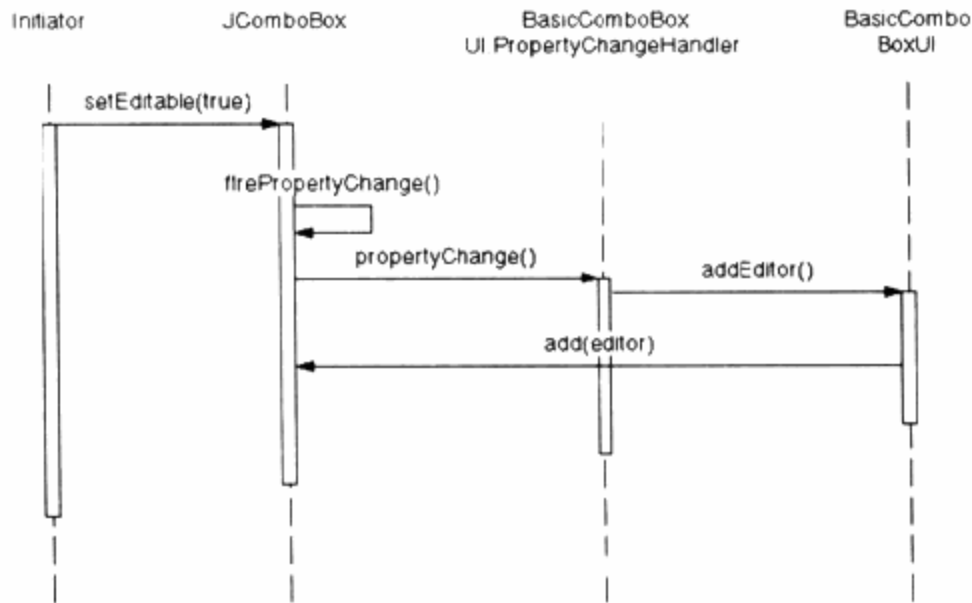


图 18-10 把一个编辑器组件添加到组合框

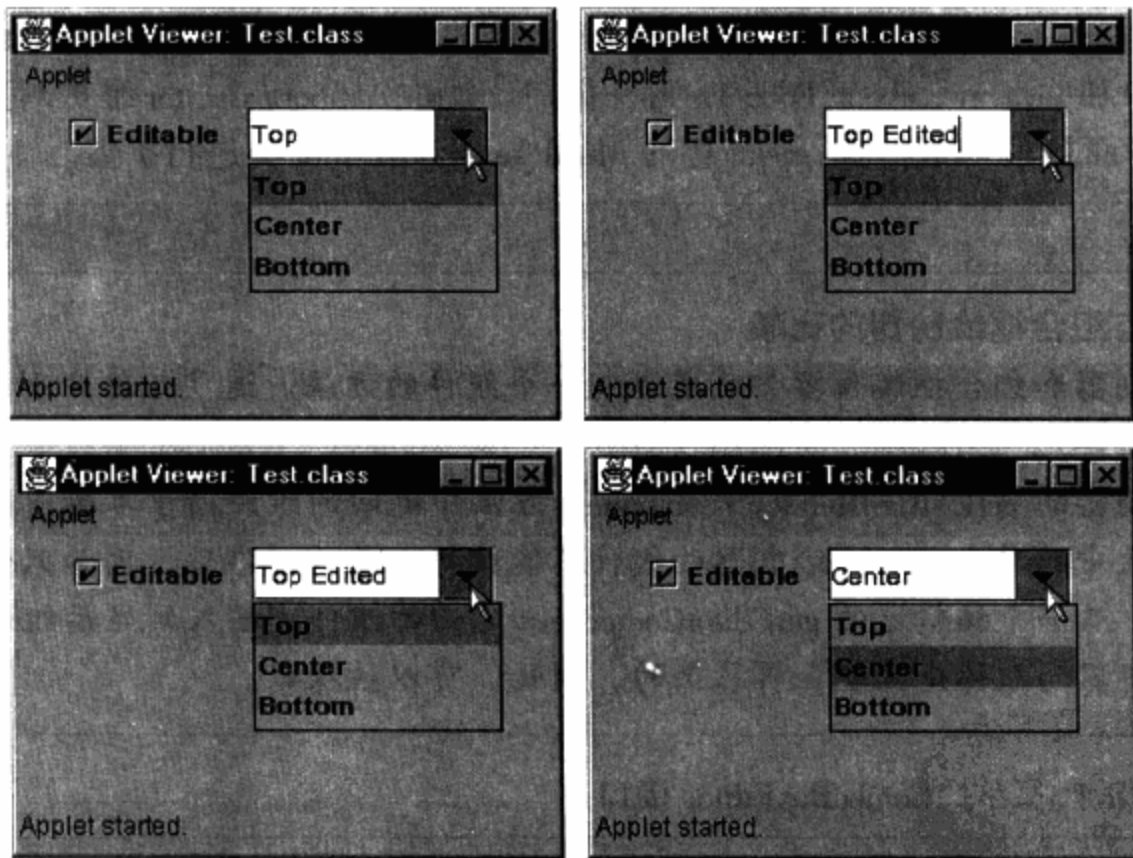


图 18-11 用 `BasicComboBoxEditor` 编辑一个项

注意，编辑没有影响在组合框下拉列表中显示的项。编辑只应用于所选取的项上，而且当在列表中又进行了选取后，将失去编辑变化。然而，定制组合框编辑器没有遵守相同的规则。例如，一个编辑器可能把一个已编辑的项添加到组合框的列表中。

如果没有显式地指定一个编辑器，则 `JComboBox` 的实例配备一个 `swing.plaf.basic` 包中的 `BasicComboBoxEditor` 的实例。`BasicComboBoxEditor` 实现 `ComboBoxEditor` 接口并提供一个用于编辑的文本域。`Swing` 没有提供 `swing` 包中 `ComboBoxEditor` 接口的一个实现。

图 18-12 示出了组合框编辑器的类图。

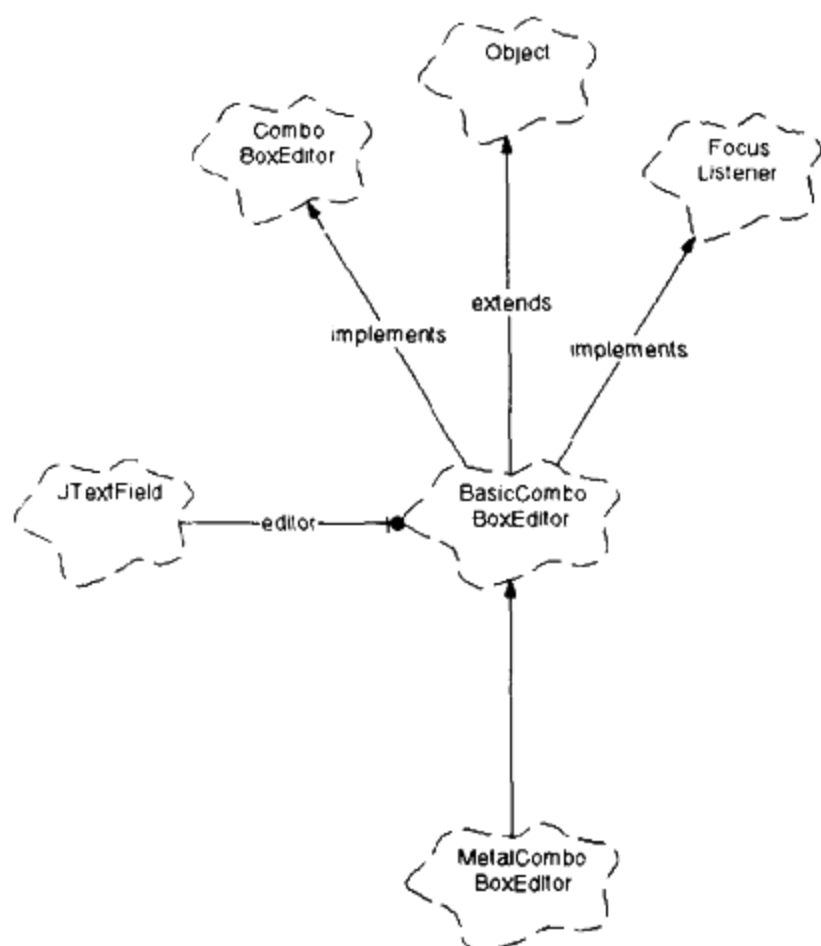


图 18-12 组合框编辑器的类图

由 `ComboBoxEditor` 接口来定义组合框编辑器的功能。`BasicComboBoxEditor` 类实现 `ComboBoxEditor` 接口和 `java.awt.event.FocusListener` 接口。`BasicComboBoxEditor` 维护对其文本域的一个 `protected` 引用，而且由 `swing.plaf.metal` 包的 `MetalComboBoxEditor` 类进行扩展。

Swing 提示

组合框编辑器与组合框绘制器的比较

组合框绘制器和组合框编辑器都实现返回一个组件的方法。这些组件分别用于（毫不惊奇地）绘制和编辑单元。

从一个绘制器的 `getListCellRendererComponent` 方法中返回的组件用于像橡皮图章那样绘制在组合框下拉列表中的单元。一个绘制器的组件不能被操纵或响应事件，因为只使用了组件的可见代表。相反，从一个编辑器的 `getEditorComponent` 返回的组件可插入到显示组合框所选项的区域中。一个编辑器的组件是呈现在屏幕上的，因此，可以被操纵。

接口总结 18-4 总结了 `ComboBoxEditor` 接口。

接口总结 18-4 `ComboBoxEditor`

1. 动作监听器登记

```

public abstract void addActionListener (ActionListener)
public abstract void removeActionListener (ActionListener)

```

组合框编辑器在已添加一个项后应该激发一个动作事件，因此，必须提供添加和删除动作监听器的方法。`BasicComboBoxEditor` 向它的文本域添加监听器并从它的文本域中删除监听器。当按下回车键而文本域又有焦点时，将激发动作事件。

2. 编辑器组件

```
public abstract Component getEditorComponent ()
public abstract void selectAll ()
```

对 `JComboBox.setEditable(true)` 的调用最终导致对编辑器 `getEditorComponent()` 方法的调用。返回的组件可插入到所选取的项所占的区域中。

`selectAll` 方法应该选取所有正在编辑的项。通过为它的编辑器组件调用 `JTextField.selectAll()`, `BasicComboBoxEditor` 实现了 `selectAll` 方法。然而, 不能从 swing 中的任何地方调用 `ComboBoxEditor.selectAll()`。

3. 项

```
public abstract Object getItem ()
public abstract void setItem (Object)
```

`setItem` 和 `getItem` 方法处理存储在组合框模型中的对象。例如, 如果一个可编辑组合框存储字符串 (代表它的模型中的颜色) 而且有一个绘制器 (在一个标签中绘制颜色图标), 则这个编辑器的 `setItem` 方法将传送模型中的一个字符串, 而不是一个图标或一个标签。

图 18-13 所示的小应用程序包含一个代表一个颜色列表的组合框。这个组合框配备了一个定制编辑器, 它允许编辑所选取的颜色。在选取项上单击鼠标将引起编辑器显示一个颜色选取器, 从这个颜色选取器中可以选取一种颜色。

最上面的图片显示这个小应用程序开始的样子, 中间的图片显示从颜色选取器中选取一种颜色, 最底下的图片显示编辑选取项后的小应用程序的样子。

这个小应用程序创建了一个带有 `Object` 数组的组合框, 每个数组项都包含一个颜色和一个标识这个颜色的字符串。设置组合框的绘制器为 `ColorRenderer` 的一个实例, `ColorRenderer` 类扩展 `JLabel` 并显示一个颜色图标和一个字符串。

这个小应用程序还创建了一个 `ColorComboBoxEditor` 实例并把它指定为组合框的编辑器。因为缺省时组合框是不可编辑的, 所以 `JComboBox.setEditable(true)` 由这个小应用程序的 `init` 方法所调用。

```
public class Test extends JApplet {
    private JComboBox comboBox =
        new JComboBox (new Object [] {
            new (new Object [] {Color.gray, "gray"}),
            new Object [] {Color.orange, "orange"}),
            new Object [] {Color.red, "red"}),
            new Object [] {Color.blue, "blue"}),
            new Object [] {Color.yellow, "yellow"}),
```

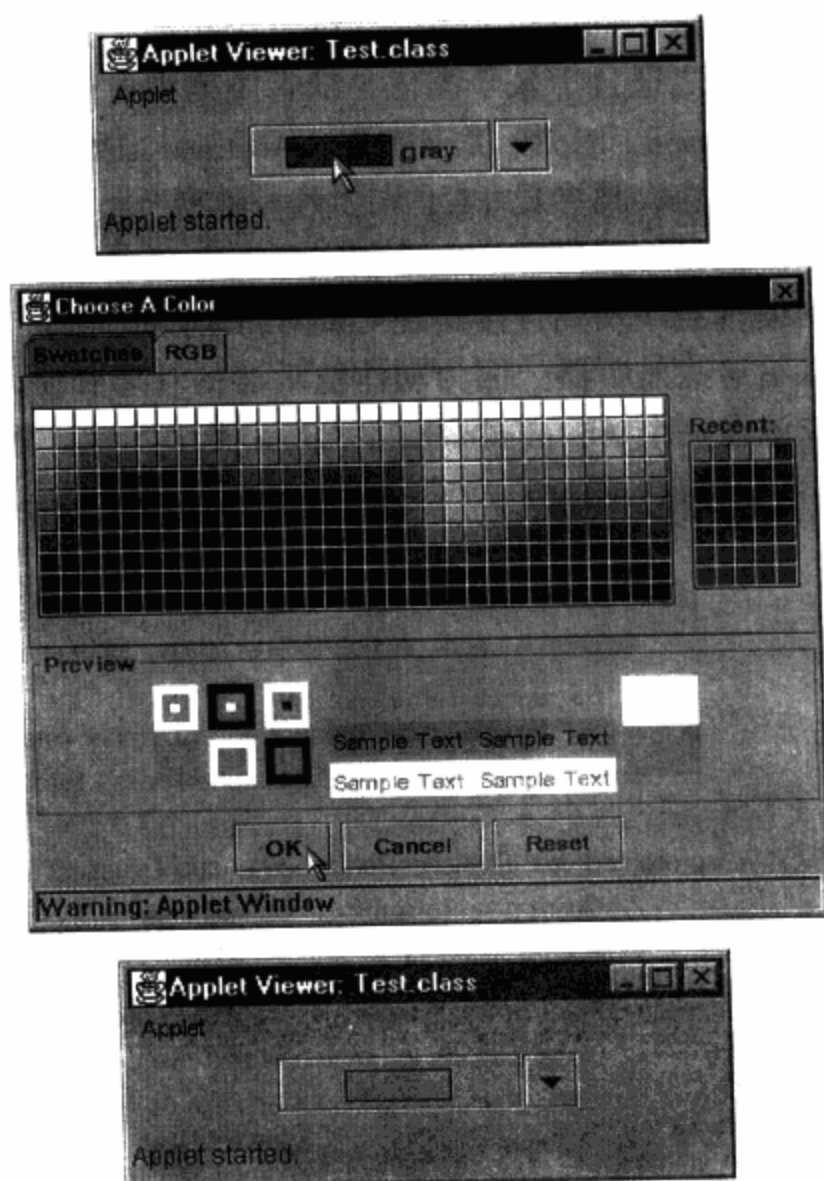


图 18-13 一个定制的组合框编辑器


```

        new Object [] {Color.magenta, "magenta"}),
        new Object [] {Color.black, "black"}),
        new Object [] {Color.green, "green"}),
        new Object [] {Color.lightGray, "lightGray"}),
        new Object [] {Color.white, "white"}),
    });

    public void init () {
        Container contentPane = getContentPane ();

        comboBox.setRenderer (new ColorRendererer ());
        comboBox.setEditor (new ColorComboBoxEditor ());
        comboBox.setEditable (true);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (comboBox);
    }
}

```

Swing 使用的一个常用设计规则是用一个接口来使对象分层，这个接口用一个抽象类来实现，而这个抽象类又由一个缺省类扩展。接口定义基本的行为，抽象类通常实现不常变化的功能，使之适合于被子类化，缺省类则为可能被定制的功能实现缺省行为。例如，ListModel 接口由 AbstractListModel 类来实现，AbstractListModel 类又被 DefaultListModel 类扩展。有关列表模型的更多信息，请参见 17.1 节“列表模型”。

但是，接口——>抽象类——>缺省类这种常用模式未用于组合框编辑器。因此，实现定制编辑器涉及到直接实现 ComboBoxEditor 接口，它包括管理动作监听器的列表和把动作事件发送给这些监听器。因为动作监听器的管理和激发动作事件是一些要重复使用的功能，所以它们应该在一个 AbstractComboBoxEditor 类中实现，如下面给出的一个 AbstractComboBoxEditor 实现：

```

abstract class AbstractComboBoxEditor implements ComboBoxEditor {
    EventListenerList listenerList = new EventListenerList ();

    public void addActionListener (ActionListener listener) {
        listenerList.add (ActionListener.class, listener);
    }

    public void removeActionListener (ActionListener listener) {
        listenerList.remove (ActionListener.class, listener);
    }

    protected void fireActionPerformed (ActionEvent e) {
        // Guaranteed to return a non-null array
        Object [] listeners = listenerList.getListenerList ();

        // Process the listeners last to first, notifying
        // those that are interested in this event
        for (int i = listeners.length-2; i >= 0; i -= 2) {
            if (listeners [i] == ActionListener.class) {
                ((ActionListener)
                    listeners [i+1]).actionPerformed (e);
            }
        }
    }
}

```

ColorComboBoxEditor 类扩展上面所列的 AbstractComboBoxEditor 类。ColorComboBoxEditor 使用一个标签来显示颜色图标和文本。颜色图标是 ColorIcon 类的一个实例，它设置 Icon 接口并绘制一个带黑边框的填充矩形，例 18-6 中列出了 ColorIcon 类。

调用 `JColorChooser.createDialog()` 来把一个颜色选取器实例化并把它包裹在一个对话框中。有关 `JColorChooser` 类的更多信息, 请参见 16.2 节“`JColorChooser`”。

```
class ColorComboBoxEditor extends AbstractComboBoxEditor {
    ColorIcon editorIcon = new ColorIcon ();
    JLabel editorLabel = new JLabel (editorIcon);

    Object [] comboBoxItem;

    JColorChooser colorChooser = new JColorChooser ();
    ActionListener okListener = new OKListener ();
    Dialog dialog = JColorChooser.createDialog (
        null, // parentComponent
        "Choose A Color", // title
        true, // modal
        colorChooser,
        okListener,
        null); // cancel listener
    ...
}
```

当构造 `ColorComboBoxEditor` 时, 鼠标监听器将添加到这个编辑器的标签中, 以便在鼠标按下时显示这个对话框。这个标签也是从这个编辑器的 `getEditorComponent` 方法中返回的组件。

```
...
public ColorComboBoxEditor () {
    editorLabel.setBorder (BorderFactory.createEtchedBorder ());

    editorLabel.addMouseListener (new MouseAdapter () {
        public void mousePressed (MouseEvent e) {
            dialog.setVisible (true);
        }
    });
}

public Component getEditorComponent () {
    return editorLabel;
}
...
}
```

当构造颜色选取器时, `OKListener` 的一个实例指定作为监听器, 当激活颜色选取器中的 `OK` 按钮时, 这个监听器将得到通知。

回忆一下, 指定组合框所包含的项为一个 `Object` 数组, 这个数组包含一个颜色和一个字符串。`OK` 监听器使用从颜色选取器中选取的颜色和一个 `null` 字符串来构造一个 `Object` 数组。这个数组是从 `getItem` 方法返回的。这个监听器还把一个动作事件激发给所有已登记的动作监听器, 正如编辑一个项时所需要的那样。

```
...
class OKListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        comboBoxItem =
            new Object [] {colorChooser.getColor (), null};

        fireActionPerformed (e);
    }
}
...
}
```

这个编辑器实现由此接口定义的 `setItem`、`getItem` 和 `selectAll` 方法。

`setItem` 方法把项存储在编辑器的 `ComboBoxEditor` 成员变量中, 并设置图标的颜色和标签的

文本。getItem 方法简单地返回项。因为这个编辑器不显示一个可选取的项，所以这里实现的 selectAll 方法为无操作。

```

    public Object getItem () {
        return comboBoxItem;
    }

    public void setItem (Object item) {
        comboBoxItem = (Object []) itemToSet;

        editorIcon.setColor ( (Color) comboBoxItem [0]);
        editorLabel.setText ( (String) comboBoxItem [1]);
    }

    public void selectAll () {
        // from ComboBoxModel interface: nothing to select
    }
}

```

例 18-6 完整地列出了图 18-3 所示小应用程序的代码。

例 18-6 实现一个定制的组合框编辑器

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class Test extends JApplet {
    private JComboBox comboBox =
        new JComboBox (new Object [] {
            new Object [] {Color.gray, "gray"}),
            new Object [] {Color.orange, "orange"}),
            new Object [] {Color.red, "red"}),
            new Object [] {Color.blue, "blue"}),
            new Object [] {Color.yellow, "yellow"}),
            new Object [] {Color.magenta, "magenta"}),
            new Object [] {Color.black, "black"}),
            new Object [] {Color.green, "green"}),
            new Object [] {Color.lightGray, "lightGray"}),
            new Object [] {Color.white, "white"}),
        });

    public void init () {
        Container contentPane = getContentPane ();

        comboBox.setRenderer (new ColorRendererer ());
        comboBox.setEditor (new ColorComboBoxEditor ());
        comboBox.setEditable (true);

        contentPane.setLayout (new FlowLayout ());
        contentPane.add (comboBox);
    }

    class ColorComboBoxEditor extends AbstractComboBoxEditor {
        ColorIcon editorIcon = new ColorIcon ();
        JLabel editorLabel = new JLabel (editorIcon);

        Object [] comboBoxItem;
    }
}

```

```

JColorChooser colorChooser = new JColorChooser ();
ActionListener okListener = new OKListener ();
Dialog dialog = JColorChooser.createDialog (
    null,          // parentComponent
    "Choose A Color", // title
    true,          // modal
    colorChooser,
    okListener,
    null);         // cancel listener

public ColorComboBoxEditor () {
    editorLabel.setBorder ( BorderFactory.createEtchedBorder () );
    editorLabel.addMouseListener ( new MouseAdapter () {
        public void mousePressed ( MouseEvent e ) {
            dialog.setVisible ( true );
        }
    });
}

class OKListener implements ActionListener {
    public void actionPerformed ( ActionEvent e ) {
        comboBoxItem =
            new Object [] { colorChooser.getColor (), null };
        fireActionPerformed ( e );
    }
}

public Component getEditorComponent () {
    return editorLabel;
}

public Object getItem () {
    return comboBoxItem;
}

public void selectAll () {
    //from ComboBoxModel interface: nothing to select
}

public void setItem ( Object item ) {
    comboBoxItem = ( Object [] ) itemToSet;
    editorIcon.setColor ( ( Color ) comboBoxItem [ 0 ] );
    editorLabel.setText ( ( String ) comboBoxItem [ 1 ] );
}

abstract class AbstractComboBoxEditor implements ComboBoxEditor {
    EventListenerList listenerList = new EventListenerList ();

    public void addActionListener ( ActionListener listener ) {
        listenerList.add ( ActionListener.class, listener );
    }

    public void removeActionListener ( ActionListener listener ) {
        listenerList.remove ( ActionListener.class, listener );
    }

    protected void fireActionPerformed ( ActionEvent e ) {
        // Guaranteed to return a non-null array
        Object [] listeners = listenerList.getListenerList ();
        // Process the listeners last to first, notifying
        // those that are interested in this event
    }
}

```

```

        for (int i = listeners.length-2; i >= 0; i = 2) {
            if (listeners [i] == ActionListener.class) {
                ((ActionListener)
                    listeners [i+1]).actionPerformed (e);
            }
        }
    }

class ColorRenderrer extends JLabel
    implements ListCellRenderer {
    private ColorIcon icon = new ColorIcon ();
    public ColorRenderrer () {
        setOpaque (true);
        setIcon (icon);
    }
    public Component getListCellRendererComponent (
        JList list,
        Object value,
        int index,
        boolean isSelected,
        boolean cellHasFocus) {
        object [] array = (object []) value;
        icon.setColor ( (Color) array [ ()]);
        setText (String) array [1];
        if (isSelected) {
            setForeground (list.getSelectionForeground ());
            setBackground (list.getSelectionBackground ());
        }
        else {
            setForeground (list.getForeground ());
            setBackground (list.getBackground ());
        }
        return this;
    }
}

class ColorIcon implements Icon {
    private Color color;
    private int w, h;
    public ColorIcon () {
        this (Color.gray, 50, 15);
    }
    public ColorIcon (Color color, int w, int h) {
        this.color = color;
        this.w = w;
        this.h = h;
    }
    public void paintIcon (Component c, Graphics g, int x, int y) {
        g.setColor (Color.black);
        g.drawRect (x, y, w-1, h-1);
        g.setColor (color);
        g.fillRect (x+1, y+1, w-2, h-2);
    }
}

```

```
public Color getColor () {
    return color;
}

public void setColor (Color color) {
    this.color = color;
}

public int getIconWidth () {
    return w;
}

public int getIconHeight () {
    return h;
}
```

组件总结 18-1 总结了 JComboBox 类。

组件总结 18-1 JComboBox

- 模型: JComboBoxModel
- UI 代表: javax.swing.plaf.basic.BasicComboBoxUI
- 绘制器: javax.swing.plaf.basic.BasicComboBoxRenderer
- 编辑器: ComboBoxEditor
- 激发的事件: ActionEvents、ItemEvents、PropertyChangeEvents
- 替换: java.awt.Choice
- 类图:

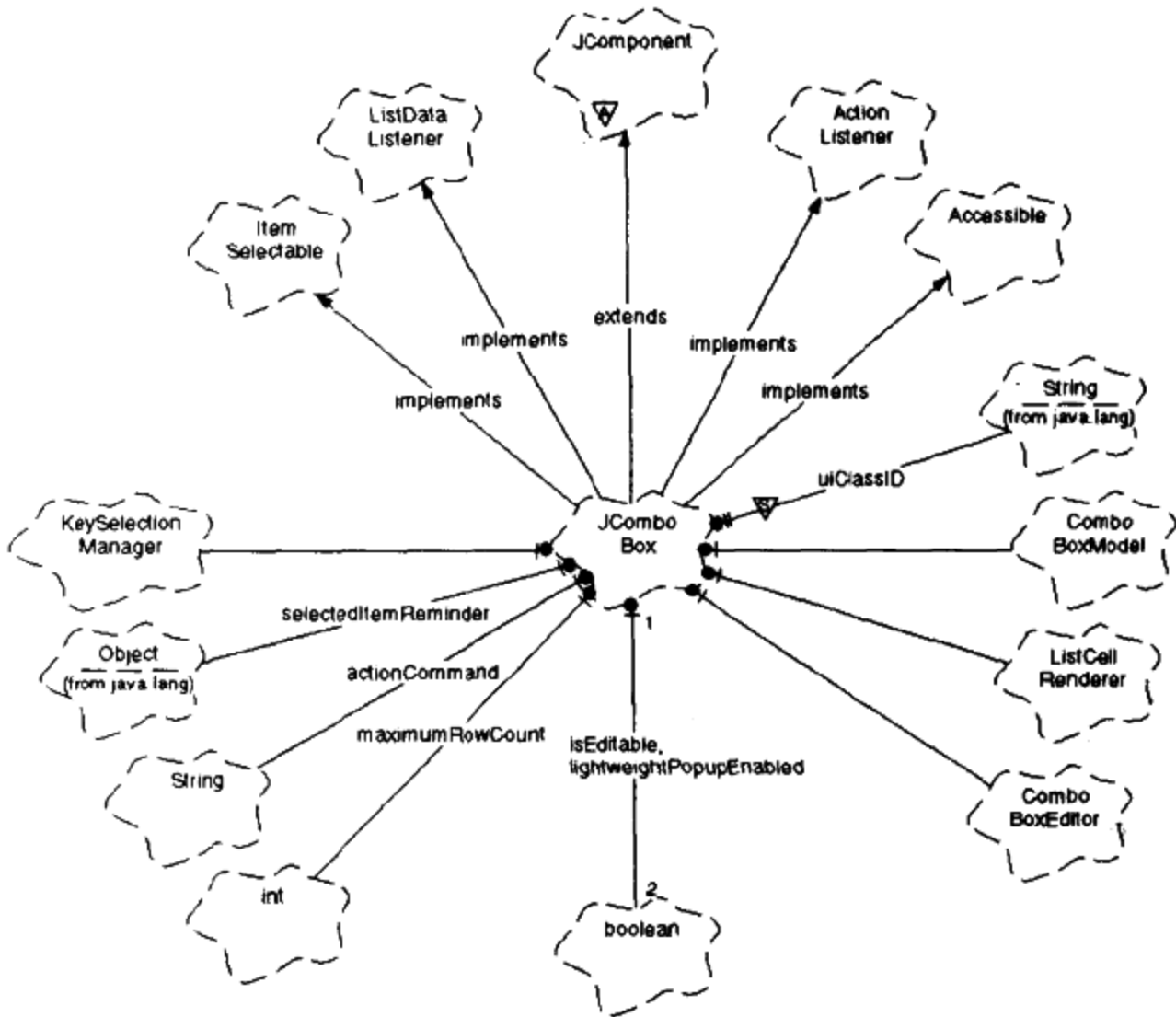


图 18-14 JComboBox 类图

与所有的轻量 Swing 组件一样，JComboBox 扩展 JComponent，并实现 Accessible 接口。

JComboBox 实现 java.awt.ItemSelectable 接口以便维护与 java.awt.Choice 的兼容，java.awt.Choice 在选取项或取消选取时将激发项事件。

当编辑一个项时，组合框编辑器把动作事件发送给已登记的动作监听器。其中的一个监听器是组合框本身，它对编辑作出反应，即更新模型的所选项并隐藏列表的弹出式菜单。因此，JComboBox 实现 ActionListener 接口。

当修改组合框的数据时，组合框模型将激发列表数据事件。JComboBox 实现 ListDataListener 接口并监听模型，以了解当组合框数据改变时是否可能重新设置所选项。

JComboBox 维护的对象说明了组合框委托给其他对象完成的任务。JComboBox 维护对它的模型、列表单元绘制器、编辑器和键选取管理器的引用。

当修改组合框的所选项时，将激发两个项事件，一个是取消以前的选取项，另一个是选取当前的选取项。对 selectedItemReminder 的 Object 引用跟踪最后的选取项，以便把它指定为取消选取事件的事件源。

动作事件使用一个字符串（称作动作命令），这个字符串标识这个动作。JComboBox 维护一个 actionCommand 字符串，缺省时它是 comboBoxChanged。

最后，JComboBox 维护一个字符串、一个 integer 和两个 boolean 引用，这些值都代表组合框的属性。

18.6.1 JComboBox 属性

表 18-3 列出了由 JComboBox 类维护的属性。

表 18-3 JComboBox 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
actionCommand	String	S	SG	"comboBoxChanged"
editable	boolean	B	S	false
editor	ComboBoxEditor	B	SG	null
itemAt	Object	L	G	null
itemCount	int	S	G	-1
keySelectionManager	KeySelectionManager	S	SG	null
lightWeightPopupEnabled	boolean	S	SG	true
maximumRowCount	int	B	SG	8
model	ComboBoxModel	B	SG	ComboBoxModel
popupVisible	Model		S	false
renderer	ListCellRenderer	B	SG	参见下面的介绍
selectedIndex	int	S	SG	-1
selectedItem	Object	S	SG	null

① B = 关联的（激发 PropertyChangeEvent）/C = 受约束的/ I = 索引的/
S = 简单的/Ch = 激发 ChangeEvent
② C = 可在创建时设置/G = 获取方法/S = 设置方法
③ L&F = 与界面样式有关

actionCommand——当选取组合框的项时，JComboBox 的实例将激发动作事件。动作事件与动作命令相关联，动作命令是一个标识动作类型的字符串。JComboBox 指定它的 actionCommand 属性作为它激发的所有动作事件的动作命令。

editable——一个 boolean 变量，它跟踪 JComboBox 的实例是否是可编辑的。有关组合框编

辑的详细内容, 请参见 18.6 节“组合框编辑器”。

itemAt——一个只读模型属性, 它代表给定索引的一个对象。

itemCount——一个只读模型属性, 它代表组合框下拉列表所包含的项数。

keySelectionManager——一个对象, 它实现 `JComboBox.KeySelectionManager` 接口。有关组合框和键选取管理的详细内容, 请参见 18.5 节“组合框键选取管理器”。

lightWeightPopupEnabled——组合框在一个弹出式菜单中显示它们的列表。缺省情况下, 弹出式菜单是一个轻量菜单, 但是有时需要使用重量菜单。因此, 可以把 `LightWeightPopupEnabled` 属性设置为 `false`, 以强制使用重量弹出式菜单。

maximumRowCount——对显示在组合框下拉列表中的项数没有限制。然而, 列表中的可见行数可以用 `maximumRowCount` 属性来控制, 这个属性的缺省值为 8。

model——`ComboBoxModel` 接口的一个实现, 它扩展 `ListModel`。

popupVisible——一个 `boolean` 属性, 它控制包含一个组合框列表的弹出式菜单的可见性。

renderer——绘制器通过实现 `ListCellRenderer` 接口来绘制列表单元。

selectedIndex——代表组合框列表中选择项的索引。

selectedItem——当前在组合框中选取的对象。

18.6.2 JComboBox 事件

当组合框的选取项变化(选取或编辑)时, `JComboBox` 的实例将激发两个项事件, 一个取消以前的选取项, 另一个选取当前的选取项。在这些项事件被激发后, 还将激发一个动作事件。

图 18-15 图示了在组合框的选取索引被修改时发生的事件序列。

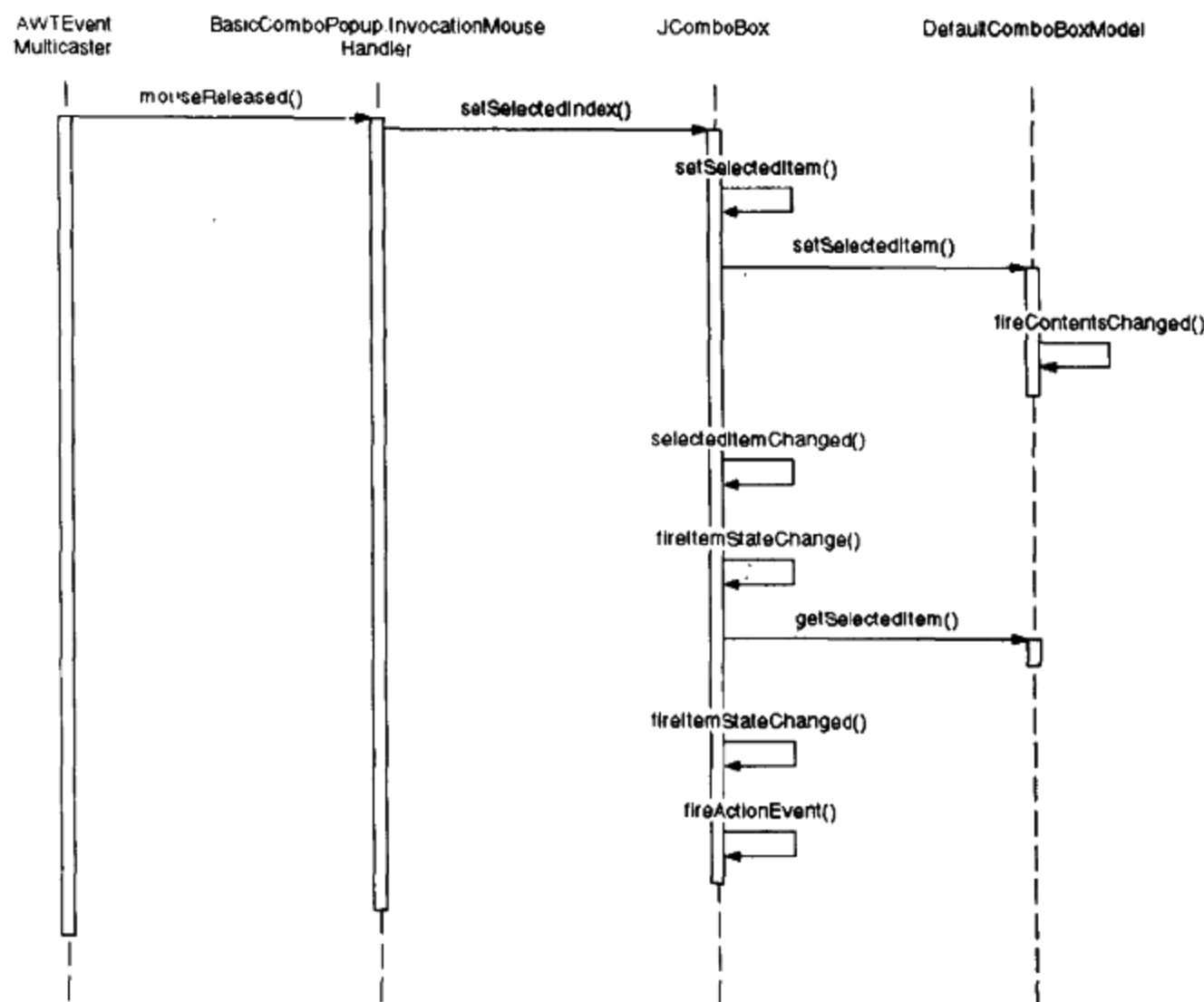


图 18-15 JComboBox 事件

当被编辑一个项后，组合框编辑器立刻激发动作事件。动作监听器可以被直接添加到一个处理编辑事件的组合框编辑器中。

1. 选取事件

图 18-16 所示的小应用程序把一个项监听器添加到它的组合框中。这个监听器显示一个消息对话框，它显示有关选取或取消选取的信息。

最上面的图片显示这个小应用程序的初始状态。中间的图片是选取了 Center 项后小应用程序的样子，中间的这些图片显示了连续的对话框，这些对话框显示在选取发生变化时激发的每个项事件。最底下的图片显示选取了 Center 项时的组合框。

例 18-7 列出了图 18-16 所示的小应用程序。

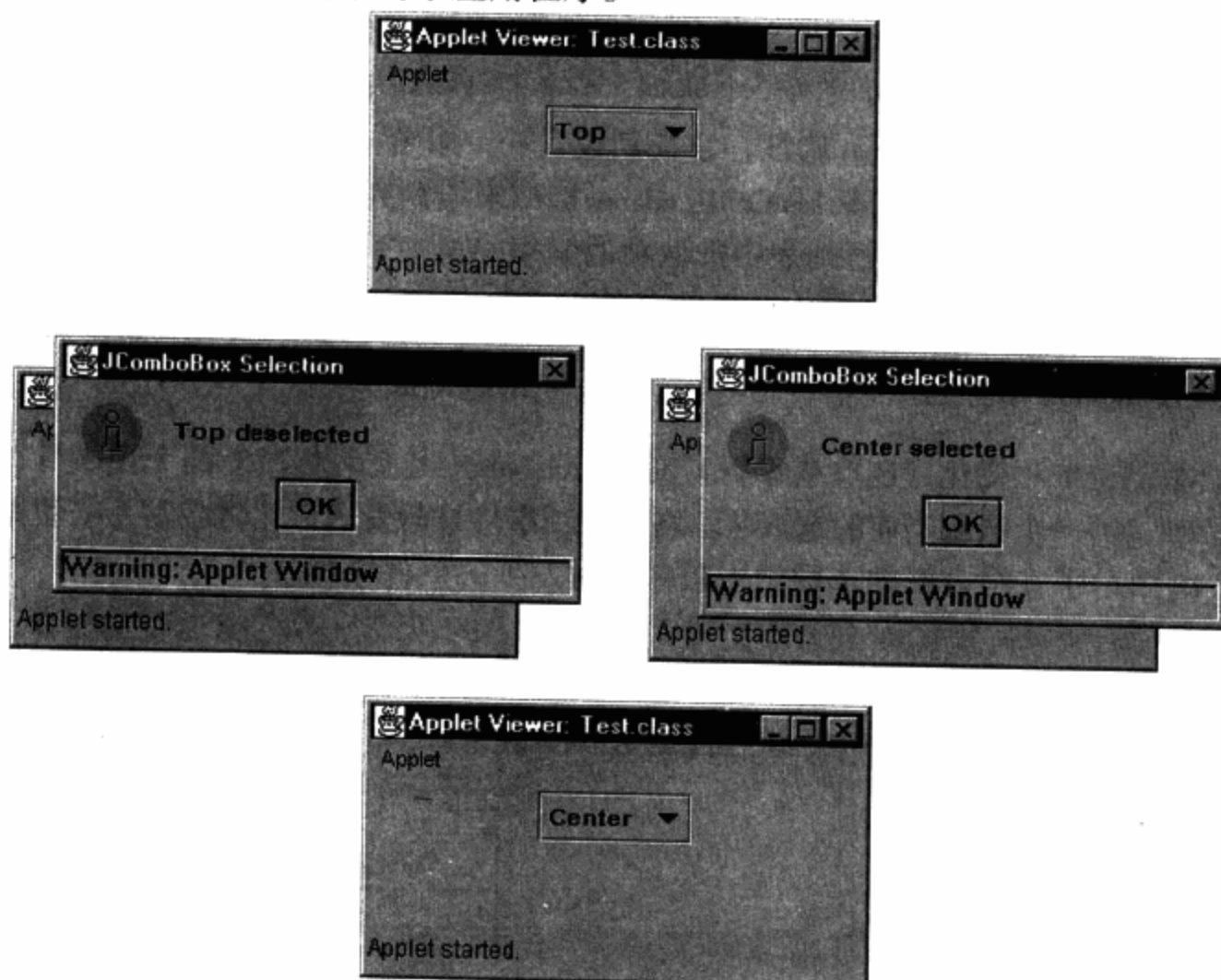


图 18-16 处理选取事件

例 18-7 处理选取事件

```
import java.applet.Applet
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JComboBox comboBox = new JComboBox ();

    public void init () {
        Container contentPane = getContentPane ();

        comboBox.addItem ("Top");
        comboBox.addItem ("Center");
        comboBox.addItem ("Bottom");

        contentPane.setLayout (new FlowLayout ());
```

```

contentPane.add (comboBox);

comboBox.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        int state = event.getStateChange ();
        String item = (String) event.getItem (); s;

        if (event.getStateChange () == ItemEvent.SELECTED)
            s = " selected";
        else
            s = " deselected";

        JOptionPane.showMessageDialog (
            comboBox, // parent component
            item + s, // message
            "JComboBox Selection", // title
            JOptionPane.INFORMATION-MESSAGE); // type
    }
});
}

```

这个小应用程序创建一个组合框，把三个项添加到这个组合框中，并把这个组合框添加到它的内容窗格中。

添加显示消息对话框（传送选取或取消选取的项）的项监听器到这个组合框中。通过调用事件的 `getStateChange`，监听器获得这个事件的状态（`ItemEvent.SELECTED` 或 `ItemEvent.DESELECTED`）。通过调用事件的 `getItem` 来获得项本身，消息对话框中要使用这个状态和这个项字符串。

2. 编辑事件

在编辑当前所显示的值时，组合框编辑器将激发动作事件。图 18-17 示出的小应用程序包含一个组合框，这个小应用程序还把一个动作监听器添加到组合框的编辑器中。当编辑组合框中显示的值时，这个监听器就更新这个小应用程序的状态区。

例 18-8 列出了图 18-17 所示小应用程序的代码。

例 18-8 处理编辑事件

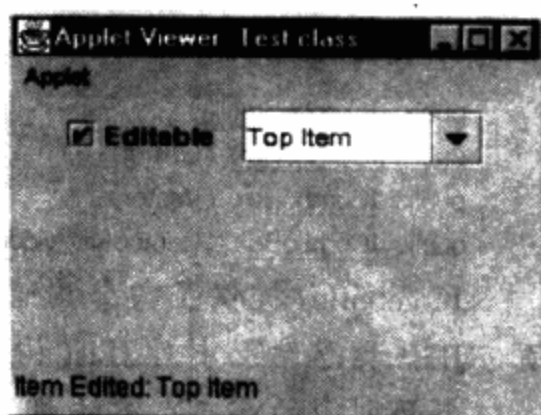


图 18-17 处理编辑事件

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Test extends JApplet {
    private JComboBox comboBox = new JComboBox ();
    private ComboBoxEditor editor = comboBox.getEditor ();

    public void init () {
        Container contentPane = getContentPane ();

        comboBox.setEditable (true);

        comboBox.addItem ("Top");
        comboBox.addItem ("Center");
        comboBox.addItem ("Bottom");
    }
}

```

```

contentPane.setLayout (new FlowLayout ());
contentPane.add (comboBox);

editor.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        String s = (String) editor.getItem ();
        showStatus ("Item Edited: " + s);
    }
});

```

通过调用组合框编辑器的 `ComboBoxEditor.getItem()` 方法, 这个监听器获得组合框显示的当前值。

18.6.3 JComboBox 类总结

类总结 18-2 列出了 `JComboBox` 的 `public` 和 `protected` 的变量和方法。

类总结 18-2 JComboBox

扩展: `JComponent`

实现: `javax.accessibility.Accessible`、`java.awt.ItemSelectable`

`java.awt.event.ActionListener`、`javax.swing.event.ListDataListener`

1. 构造方法

```

public JComboBox ()
public JComboBox (Object [])
public JComboBox (Vector)
public JComboBox (ComboBoxModel)

```

`JComboBox` 提供了四个构造方法, 它们几乎与 `JList` 类提供的构造方法相同——参见第 1062 页“`JList` 类总结”中对 `JList` 构造方法的介绍。

上面列出的前三个方法都为组合框配备了模型, 这些模型是 `DefaultComboBoxModel` 的实例。对包含在 `Object` 数组 (上面列出的第二个构造方法的参数) 中的对象的引用拷贝在一个由 `DefaultComboBoxModel` 维护的矢量中。

用上面所列的最后一个构造方法, 可以在构造时显式地指定一个特定的组合框模型。

2. 方法

(1) 由 `ActionListener`/`ItemListener`/`ListDataListener` 定义的方法

```

public void actionPerformed (ActionEvent)
public void addItemListener (ItemListener)
public void removeItemListener (ItemListener)
public Object [] getSelectedObjects ()
public void contentsChanged (ListDataEvent)
public void intervalAdded (ListDataEvent)
public void intervalRemoved (ListDataEvent)

```

`JComboBox` 的实例登记它们自己作为编辑器的动作监听器。当在一个组合框中编辑一个值时, 组合框编辑器利用它的 `actionPerformed` 方法把一个动作事件发送给这个组合框。`JComboBox.actionPerformed()` 把组合框的选取项设置为已编辑的值, 并隐藏与这个组合框相关联的弹出式菜单。

`JComboBox` 实现 `java.awt.event.ItemListener` 接口, 以便维护与 `java.awt.Choice` 的兼容性。`getSelectedObjects` 方法返回一个 `Object` 数组, 它总是只含有一项 (对组合框选取项的一个引用)。

JComboBox 的实例向它们的模型登记它们自己,把自己作为列表数据的监听器。当修改存储在组合框模型中的数据时,用上面所列的最后三种方法之一来通知相关联的组合框。JComboBox 按相同的方式来处理所有的数据修改,即在需要时更新选取项。

(2) 组合框代表: 编辑器/键选取管理器/模型/绘制器

```
public ComboBoxEditor getEditor ()
public JComboBox.KeySelectionManager getKeySelectionManager ()
public ComboBoxModel getModel ()
public List CellRenderer getRenderer ()

public void setEditor (ComboBoxEditor)
public void setKeySelectionManager (JComboBox.KeySelectionManager)
public void setModel (ComboBoxModel)
public void setRenderer (ListCellRenderer)

protected JComboBox.KeySelectionManager createDefaultKeySelectionManager ()
public void configureEditor (ComboBoxEditor editor, Object item)
public boolean selectWithKeyChar (char)
```

上面所列的前八个方法是四个组合框代表的访问方法。编辑、键选取、数据存储或获取和单元绘制都是组合框的插入式行为。

缺省的键选取管理器由 createDefaultKeySelectionManager 方法创建,该方法的名字由此得来。这是一个 protected 方法,以便缺省的键选取管理器可以由一个 JComboBox 扩展重载。

当修改组合框的内容时,将调用 configureEditor 方法,这个方法为编辑器设置项。

selectWithKeyChar 方法程序地模仿键选取。

(3) 项方法

```
public void addItem (Object)
public void insertItemAt (Object item, int index)
public void removeAllItems ()
public void removeItem (Object)
public void removeItemAt (int index)

public Object getItemAt (int index)
public int getItemCount ()

public Object getSelectedItem ()
public void setSelectedItem (Object)
protected void selectedItemChanged ()
```

上面列出的方法用于处理组合框的项。上面所列的第一组方法在面向对象设计方面有些不正常,因为这些方法不能很好地与所有的组合框一起工作。如果为具有固定模型的组合框调用这些方法,则将弹出一个异常信息。getItemAt 和 getItemCount 方法不修改数据,因此可用于易变的和不变的组合框。

上面所列的最后一组方法与选取项有关。作为在 ComboBoxModel 中定义的方法,getItemAt 和 setSelectedItem 方法有相同的原型。事实上,JComboBox 把这两个方法直接委托给它的模型。

SelectedItemChanged 方法激发两个项事件,一个用于取消以前所选的项(如果有的话),另一个用于选取当前的项。在激发项事件的同时也激发了一个动作事件。

(4) 事件/监听器方法

```
public void addActionListener (ActionListener)
public void removeActionListener (ActionListener)
public String getActionCommand ()
public void setActionCommand (String)
```

```
protected void fireActionEvent ()
protected void fireItemStateChanged (ItemEvent)
protected void installAncestorListener ()
public void processKeyEvent (KeyEvent)
```

上面所列的这些方法用来处理事件。当修改组合框所选取的项时，组合框将激发动作事件。因此，提供了添加和删除动作监听器的方法。

每个动作事件 (java.awt.event.ActionEvent) 都有一个称为动作命令的字符串，通常用它来标识动作的类型。缺省情况下，与动作事件 (由 JComboBox 的实例激发) 相关联的动作命令是 comboBoxChanged，这个动作命令是可设置的。

JComboBox 提供两个 protected 方法，它们分别把项和动作事件发送给所有已登记的项监听器和动作侦听器，这些方法可以由 JComboBox 的扩展重载。

当移动一个组件或它的父组件之一为可见或不可见时，所有的 Swing 组件将激发父组件事件。installAncestorListener 方法把一个父组件监听器添加到组合框中，这个监听器在一个父组件事件发生时将隐藏组合框的弹出式菜单。

processKeyEvent 方法在 JComponent 类中重载，以便隐藏组合框的弹出 (如果按下了 Tab 键)。

(5) 弹出式菜单方法

```
public void hidePopup ()
public void showPopup ()
public void setPopupVisible (boolean)
public boolean isPopupVisible ()
public boolean isLightWeightPopupEnabled ()
public void setLightWeightPopupEnabled (boolean)
```

调用 setLightWeightPopupEnabled (false) 将强制一个组合框使用中量或重量弹出式菜单。缺省时，组合框使用一个轻量弹出式菜单，除非确定必须使用重量弹出式菜单。例如，如果一个组合框的弹出式菜单超过了它的父窗口的边界，则一个轻量弹出式菜单将被它的窗口的边界剪切。在这种情况下，缺省，使用一个重量弹出式菜单。

然而，总有这样一种情况，即应该使用重量弹出式菜单的时候，却使用了缺省的轻量弹出式菜单。例如，如果一个弹出式菜单与一个重量组件交迭，则这个弹出式菜单将在这个重量组件下面显示。在这种情况下，就需要强制组合框使用重量弹出式菜单。

图 18-18 所示的小应用程序包含一个按钮和一个组合框。按钮上添加了一个动作监听器，以便为这个组合框调用 showPopup ()。

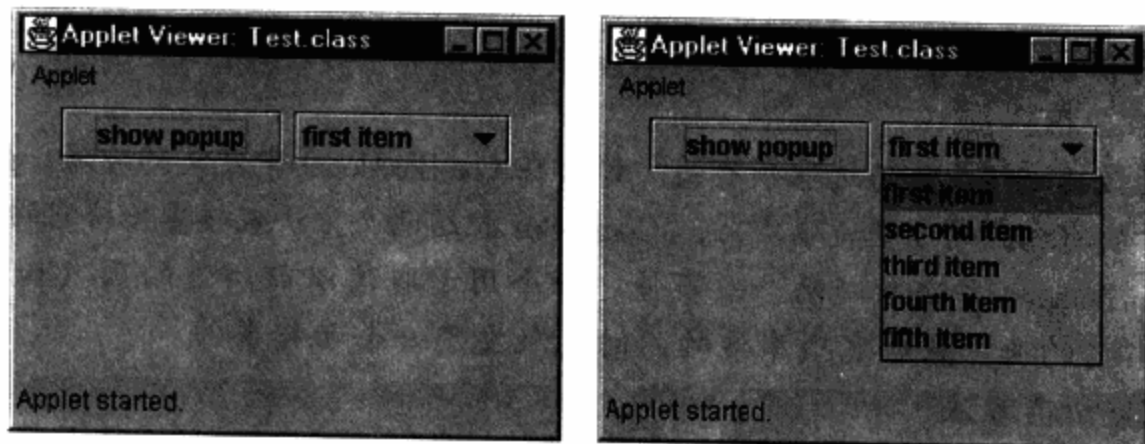


图 18-18 程序地显示一个组合框的弹出式菜单

例 18-9 列出了图 18-18 所示小应用程序的代码。

例 18-9 手工显示一个组合框的弹出式菜单

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JButton button = new JButton ("show popup");
        final JComboBox combo = new JComboBox ();

        combo.addItem ("first item");
        combo.addItem ("second item");
        combo.addItem ("third item");
        combo.addItem ("fourth item");
        combo.addItem ("fifth item");

        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                combo.showPopup ();
            }
        });
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (button);
        contentPane.add (combo);
    }
}

```

(6) 属性访问方法

```

public int getMaximumRowCount ()
public int getSelectedIndex ()

public boolean isEditable ()
public boolean isFocusTraversable ()

public void setEditable (boolean)
public void setEnabled (boolean)
public void setMaximumRowCount (int)
public void setSelectedIndex (int)

```

上面所列的这些方法是组合框属性的访问方法，组合框的这些属性在 18.6.3 节“JComboBox 类总结”中已做了介绍。

(7) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public ComboBoxUI getUI ()
public String getUIClassID ()
public void setUI (ComboBoxUI)
public void updateUI ()

```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

18.6.4 AWT 兼容

JComboBox 提供了与 java.awt.Choice 相同的基本功能。AWT Choice 组件是不可编辑的，而且只能显示字符串，而 JComboBox 的实例可以被编辑，并可以带单元绘制器和键选取管理器。

java.awt.Choice 和 JComboBox 都实现 java.awt.ItemSelectable 接口，这个接口定义下面的方法：

- void addItemListener (ItemListener)
- Object [] getSelectedObjects ()
- void removeItemListener (ItemListener)

表 18-4 列出了由 java.awt.Choice 提供的 public 方法和 JComboBox 类的对应方法。

表 18-4 public java.awt.Choice 的方法和 JComboBox 的对应方法^①

java.awt.Choice 的方法	JComboBox 的对应方法
synchronized void add (String)	void addItem (Object)
synchronized void addItem (String)	void addItem (Object)
synchronized void addItemListener (ItemListener)	void addItemListener (ItemListener)
String getItem (int)	Object getItemAt (int)
int getItemCount ()	int getItemCount ()
int getSelectedIndex ()	int getSelectedIndex ()
synchronized String getSelectedItem ()	Object getSelectedItem ()
synchronized Object [] getSelectedObjects ()	Object [] getSelectedObjects ()
synchronized void insert (String, int)	void insertItemAt (Object, int)
synchronized void remove (int)	void removeItemAt (Object, int)
synchronized void remove (String)	void removeItem (Object)
synchronized void removeAll ()	void removeAllItems ()
synchronized void removeItemListener (ItemListener)	void removeItemListener ()
synchronized void select (int)	void setSelectedIndex (int)
synchronized void select (String)	void setSelectedItem (Object)

① 斜体字指示 java.awt.Choice 方法与 JComboBox 的对应方法有相同的原型。

18.7 本章回顾

JComboBox 是一个功能强大、可配置的组件。可以为组合框编写定制的绘制器和编辑器，以便适应应用组合框的各种复杂情况。另外，组合框提供了键选取管理器。遗憾的是，JList 组件不能共享由 JComboBox 实现的键选取管理功能。

第 19 章 表 格

Swing 表格显示数据行与数据列，它是 Swing 中最复杂的组件之一。事实上，Swing 提供一个单独的包（Swing.table），这个包包含表格的支持接口和类。

Swing 表格由一个表格头部（它显示列头部）、表格列和单元值组成。表格还包含行和表格单元，但是行和表格单元不像头部和列那样是对象。Swing 表格的结构如图 19-1 所示。

表格支持许多选取模式，包括行、列和单元选取。表格单元值由实现 Table CellRenderer 接口的对象绘制并由实现 CellEditor 接口的对象进行编辑。

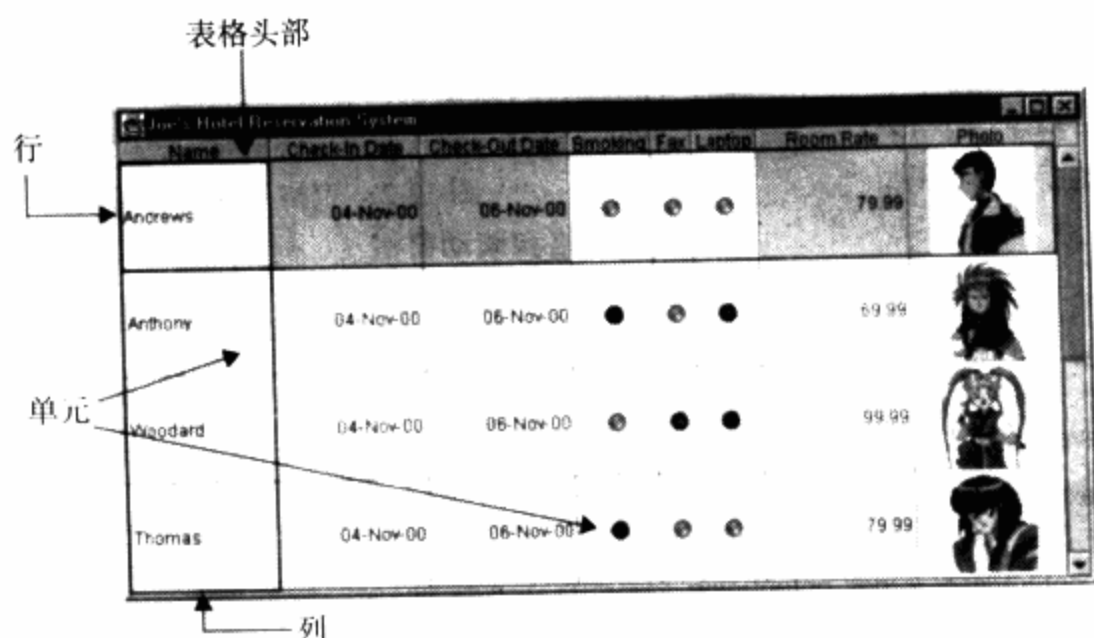


图 19-1 Swing 表格结构

19.1 表格和滚动

必须强调有关表格和滚动的两个问题：把表格放在一个滚动窗格中的效果和 Scrollable 接口的 JTable 的实现。

图 19-2 所示的应用程序包含两个相同的表格。下面的表格包含在一个滚动窗格中，而上面的表格被直接添加到这个应用程序的内容窗格中。

从图 19-2 中可以看出，放在滚动窗格中的表格与未放在滚动窗格中的表格有两点明显不同：列头部的可视性和调整表格大小的方式。包含在滚动窗格中的表格其列头部是作为这个滚动窗格的头部视图来实现的，因此，只有处在滚动窗格中的表格才能显示列头部。如果表格不包含在滚动窗格中，那么其列头部是不可见的。有关滚动窗格头部和头部视图的更多信息，请参见 13.2.1 节“滚动窗格头部”。

与不包含在滚动窗格中的表格不同，包含在滚动窗格中的表格还可以调整其大小。下面是它的原因：

JTable 实现 Scrollable 接口，要了解 Scrollable 接口的信息，请参见 13.3 节“Scrollable 接口”。GetPreferredScrollableViewportSize () 方法是由 Scrollable 接口定义的方法之一，它由 JTable 实现并返回一个缺省的大小 (450 400)^①，它表示表格视口的首选尺寸。图 19-2 所示的应用程

① 在以后的 Swing 版本中缺省值可能会变动。

序的内容窗格有一个 FlowLayout 布局管理器, 它根据组件的首选大小来调整组件的大小, 因此, 这个视口有 450 像素点宽和 400 像素高。

是 Scrollable 接口定义的另一方法 `getScrollableTracksViewportWidth()` 方法, 它由 `JTable` 来实现, 以便缺省时视口所包含的表格宽度与这个视口的宽度相同。因此, 包含在图 19-2 的滚动窗格中的表格宽度与其视口宽度相同。

图 19-2 所示的应用程序有一个简单的实现, 例 19-1 列出了应用程序的完整代码。

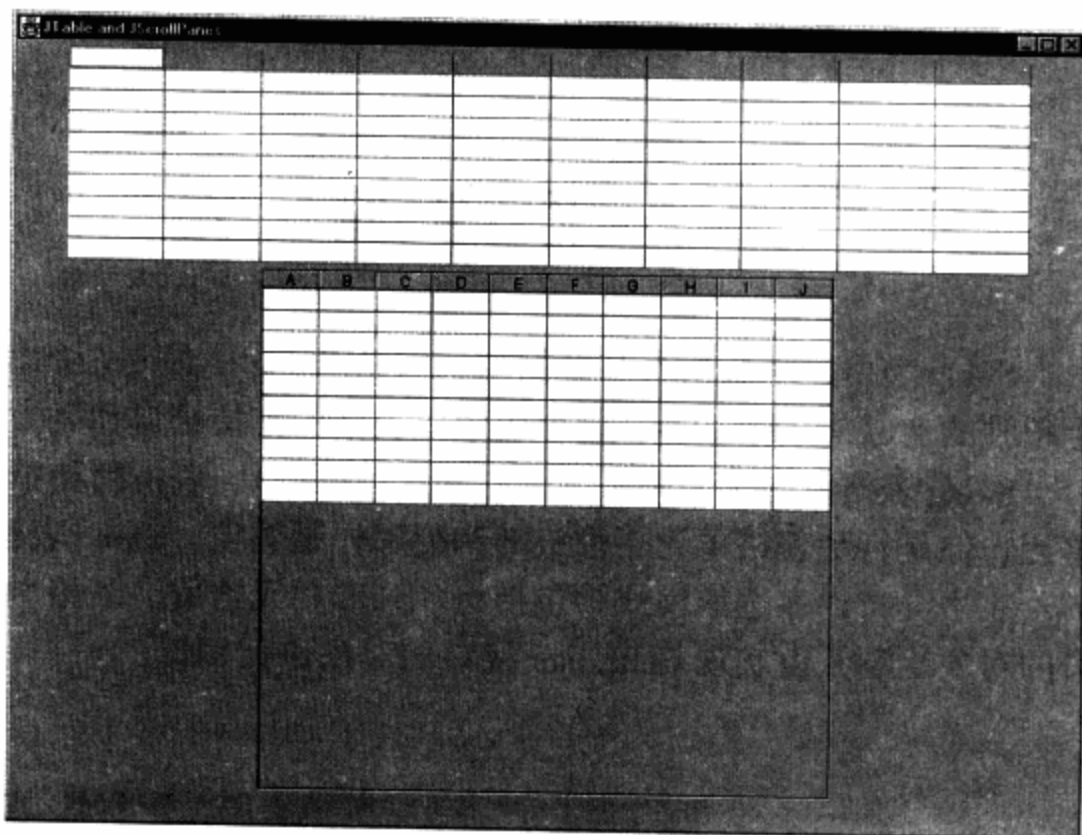


图 19-2 表格和滚动窗格

例 19-1 表格和滚动窗格

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    public Test () {
        Container contentPane = getContentPane();
        contentPane.setLayout (new FlowLayout());
        contentPane.add (new JTable (10, 10));
        contentPane.add (new JScrollPane (new JTable (10, 10)));
    }

    public static void main (String args []) {
        GJApp.launch (new Test (),
            "Tables and Scrollpanes", 100, 100, 850, 700);
    }
}
```

Swing 提示

把 JTable 的实例放在滚动窗格中

Swing 表格几乎总是包含在滚动窗格中。将表格嵌入滚动窗格的原因是, 只有包含在滚动窗格中的表格才能自动显示列头部。指定表格列头部为表格所在的滚动窗格的头部视图。因此, 如果表格不包含在滚动窗格中, 那么其列头部是不可见的。

注意 在 Swing 1.02 及以前的版本中, 必须使用静态方法 `JScrollPane.createScrollPaneForTable()` (现已取消) 来创建一个用于表格的滚动窗格。

块增量和单元增量

Scorllable 接口定义两个方法: `getScrollableBlockIncrement()` 和 `getScrollableUnitIncrement()`, 它们分别定义表格的块增量和单元增量。有关块增量和单元增量的详细内容, 请参见 13.4.2 节“块增量和单元增量”。

对于垂直滚动, 单元增量等于表格的行高, 块增量等于表格的可视行数减去一, 如图 19-3 所示。图 19-3 的上图将单元 (0, 0) 显示在表格的左上角, 中图和下图分别显示了表格向下滚动一个单元增量和一个块增量后的情况。

水平滚动的块增量和单元增量的情形如图 19-4 所示。块的水平滚动增量等于表格的可视宽度。在 Swing 1.1 FCS 中, 单元滚动增量等于 100 个像素。

19.2 表格模型

表格维护三种不同的模型: 表格模型、表格列模型和列表选取模型, 如图 19-5 所示。

表格模型实现 `TableModel` 接口并负责提供表格的单元值。

表格的列表选取模型负责表格行的选取; 而表格列模型负责列的选取, 它有自己的列表选取模型。表格列模型还维护对表格列的引用并提供添加、移动和删除列的能力。

`JTable` 类实现 `ListSelectionListener`、`TableModelListener` 和 `TableColumnModelListener` 接口来监听它的模型。通过把事件发送给表格监听器来处理表格模型和表格列模型的变化。一旦选取模型发生变化, 表格的相应区域也重新绘制。

表 19-1 总结了每种模型的功能和它们所激发的事件。

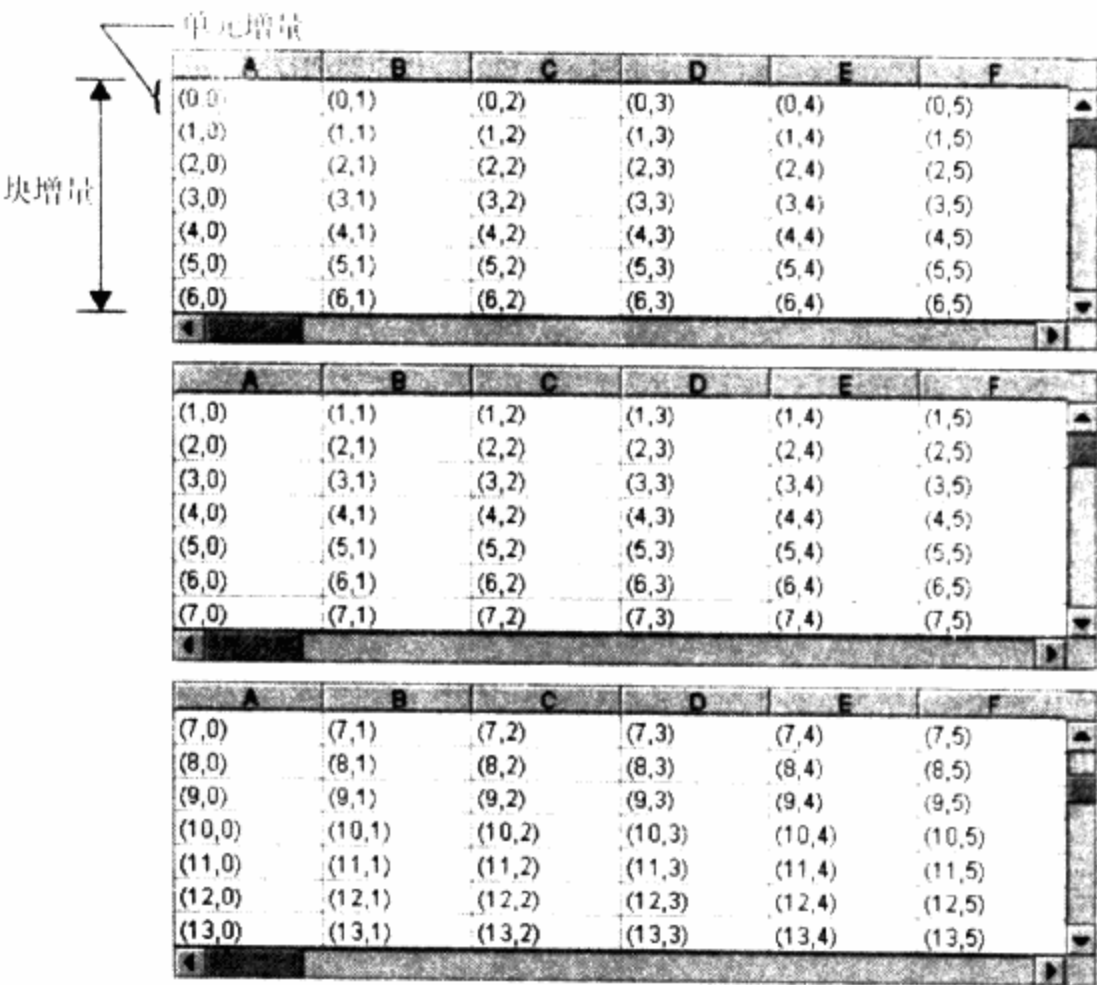


图 19-3 垂直滚动的块增量和单元增量

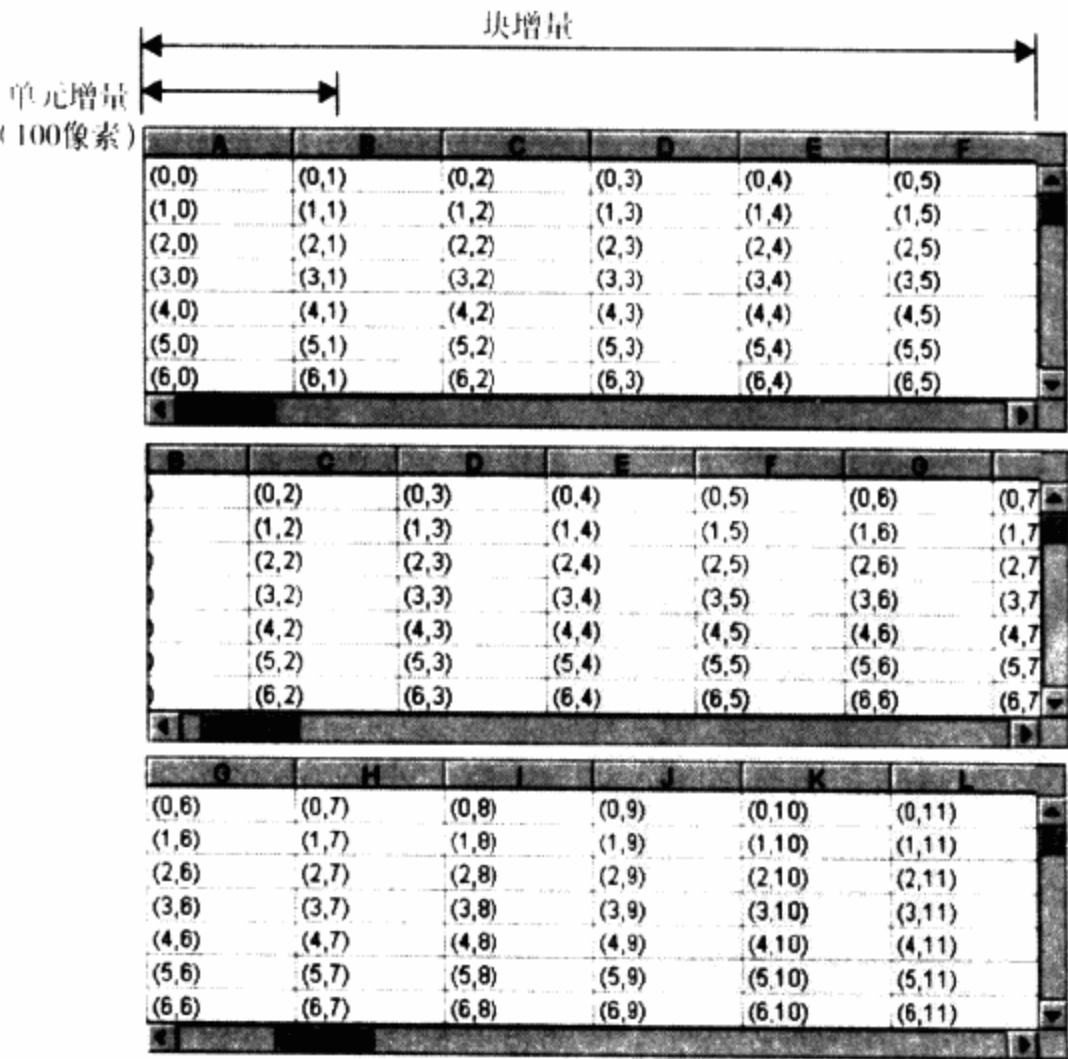


图 19-4 水平滚动的块增量和单元增量

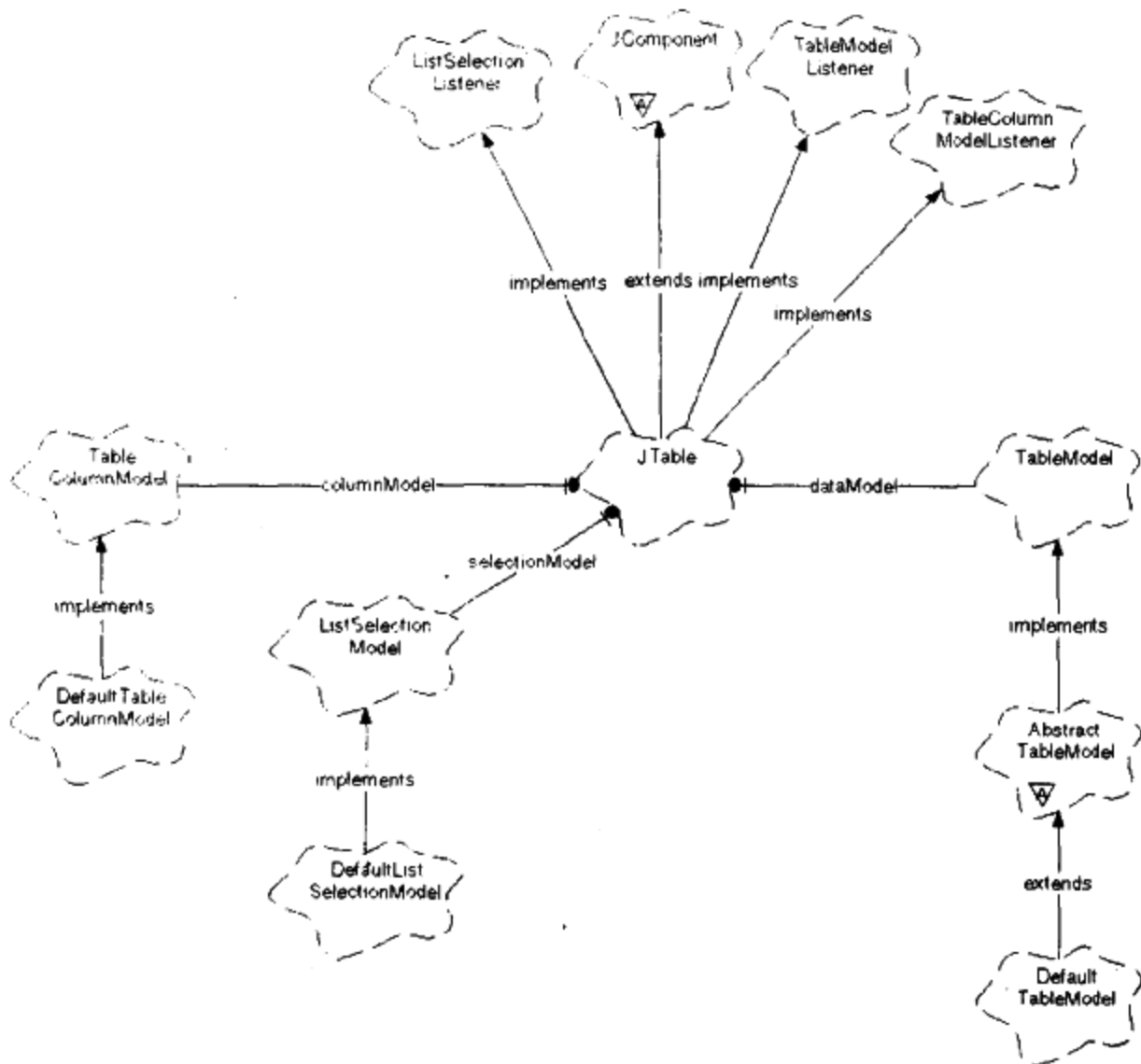


图 19-5 表格模型类图

表 19-1 表格模型

模型	功能	激发的事件
ListSelectionModel	维护行选取模式、选取间隔和一个选取是否正在调整	ListSelectionEvent
TableColumnModel	存储表格列；添加、移动和删除列；跟踪下面的列属性：选取、边距、索引、总宽度和列数	TableColumnModelEvent、ChangeEvent ^①
TableModel	为单元数据、数据类型、行数、列数及单元是否可编辑提供访问方法	TableModelEvent

① 当设置列边距时，该事件由 DefaultTableColumnModel 激发。

19.2.1 表格数据模型

JTable 不存储它的单元数据。JTable 的所有实例把它们单元值交给实现 TableModel 接口的对象来管理。

当用下面的 JTable 构造方法来构造 JTable 的一个实例时，可以指定表格单元值。

- public JTable (Object [] [] rowData, Object [] columnNames)
- public JTable (Vector rowData, Vector columnNames)
- public JTable (TableModel , TableColumnModel)
- public JTable (TableModel)

可以在构造期间把单元值指定为数据，以 Object 数组或矢量形式或作为表格和（或）表格列模型。

还可以在构造后用下列 JTable 方法来指定在表格中显示的对象：

- `public void setModel (TableModel)`
- `public void setValueAt (Object value, int row, int col)①`

表格数据模型由 TableModel 接口定义，该接口由 AbstractTableModel 类实现，而 AbstractTableModel 类扩展了 DefaultTableModel 类，如图 19-6 所示。

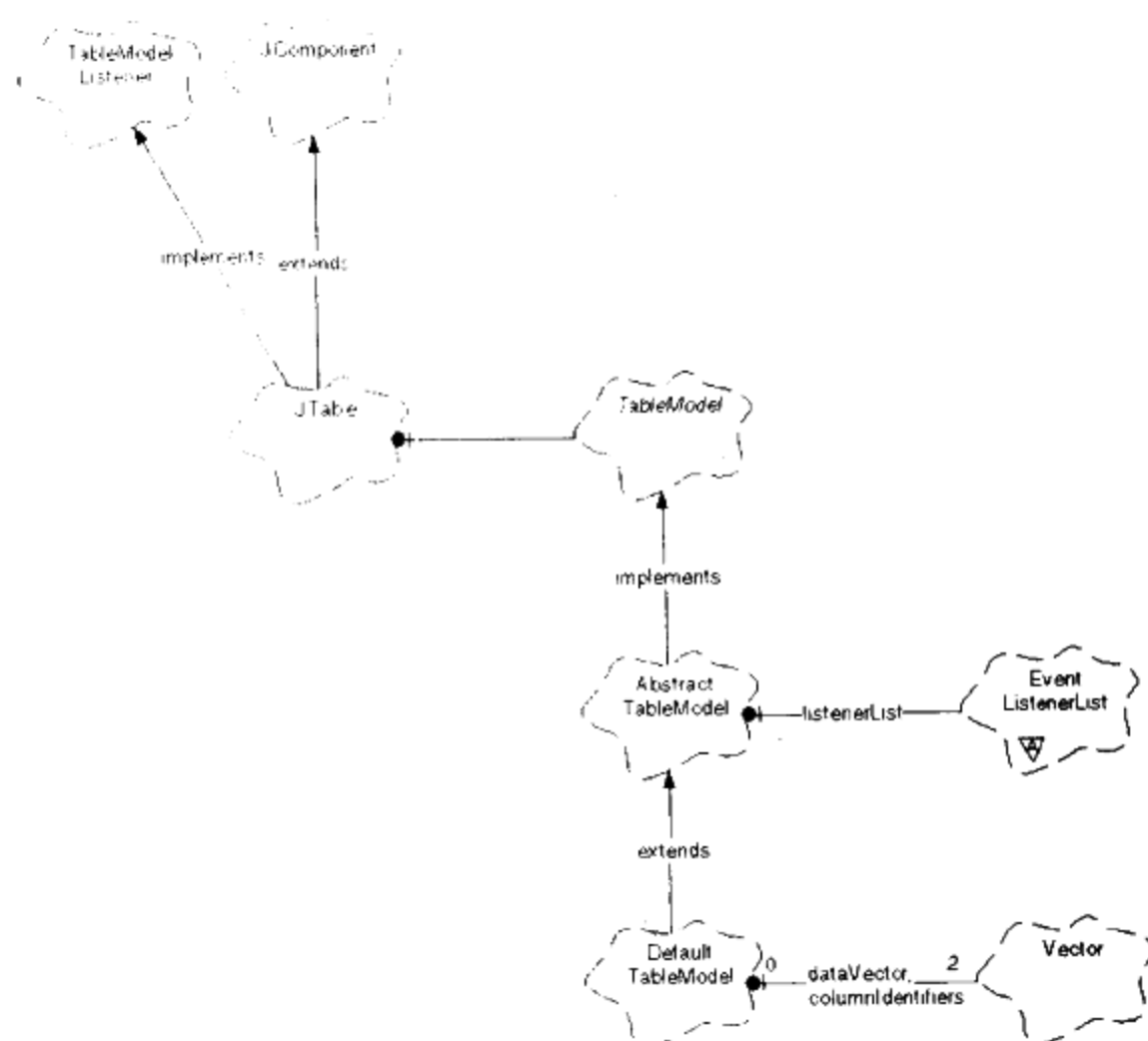


图 19-6 表格数据模型的类图

AbstractTableModel 提供缺省的模型行为并负责登记表格模型监听器，由其所包含的 EventListenerList 验证。DefaultTableModel 存储两个矢量，一个包含对列标识的引用，另一个包含表格的行数据。

TableModel 接口很少直接由开发人员实现，因为 AbstractTableModel 提供健全的缺省行为和一组方便的方法。其中，这个行为可以有选择地重载，而这些方法把事件发送给表格模型监听器。因此，大多数定制的表格模型可能扩展 AbstractTableModel 而不是直接实现 TableModel 接口。

DefaultTableModel 类除了有由 AbstractTableModel 实现的功能外，还提供两个特性：可变性和一个存储机制。DefaultTableModel 以一组矢量的形式存储单元值并提供许多修改行和列的方法。DefaultTableModel 还重载 AbstractTableModel 类中 setValueAt() 方法的空实现。

表 19-2 总结了 JTable 数据模型类。

① setValueAt 方法只有当一个表格的模型是可编辑的时候才有效。

表 19-2 表格数据模型

模型	类/接口	功能/目的	使用/实现/扩展
TableModel	接口	监听器登记、单元值访问方法、列名，类的访问方法	很少直接实现
AbstractTableModel	类	调用方法的事件	由大多数定制表格模型来扩展
DefaultTableModel	类	以一组矢量的方式存储单元值	用于小到中型的数据模型，其数据可变适于以矢量方式存储

19.2.2 TableModel 接口

TableModel 接口定义表格数据的操作方式。表格模型维护单元值，并且在单元值变化时把 TableModelEvent 事件发送给 TableModelListener 监听器。
接口总结 19-1 总结了 TableModel 接口。

接口总结 19-1 TableModel

1. TableModelListener 登记

```
public abstract void addTableModelListener (TableModelListener)
public abstract void removeTableModelListener (TableModelListener)
```

表格模型把 TableModelEvent 事件发送给 TableModelListener 监听器，因此表格模型必须跟踪已登记的监听器。

2. 单元值和可编辑性

```
public abstract Object getValueAt (int row, int column)
public abstract void setValueAt (Object value, int row, int column)
public abstract boolean isCellEditable (int row, int column)
```

将表格单元值作为 Object 的引用来访问，即任何类型的对象都可以保存在表格单元中。如果一个表格模型的 isCellEditable 返回 true，就说明表格单元是可编辑的。

3. 列和行

```
public abstract int getColumnCount ()
public abstract int getRowCount ()
public abstract Class getColumnClass (int index)
public abstract String getColumnName (int index)
```

表格模型跟踪行数和列数，用上面所列的前两种方法来访问行数和列数。

表格模型还维护存储在特定列中的对象的列名和类。

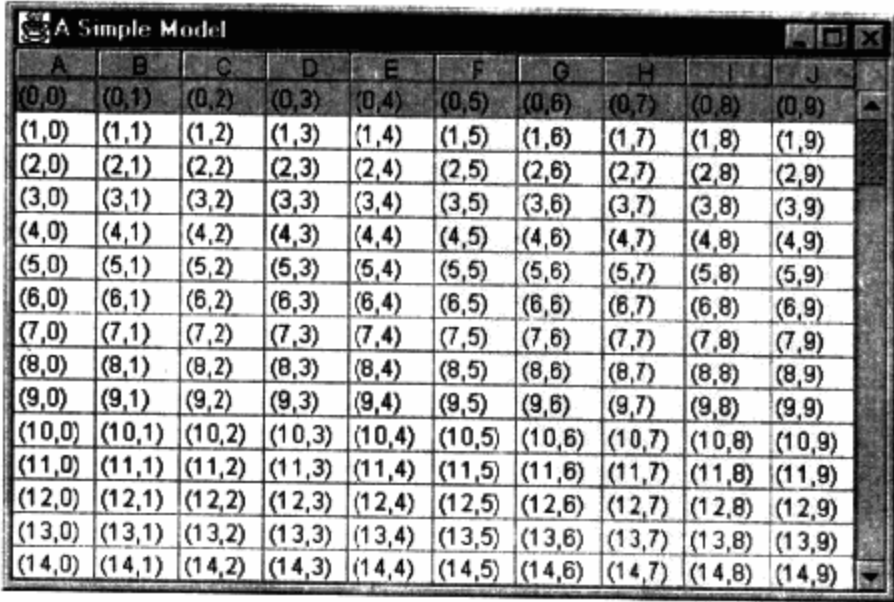


图 19-7 一个简单的表格模型

19.2.3 AbstractTableModel

AbstractTableModel 实现由 TableModel 接口定义的方法，那些返回特定数据信息的方法除外：

- public Object getValueAt (int row,

int col)

- public int getRowCount ()
- public int getColumnCount ()

AbstractTableModel 还提供七种激发 TableModelEvents 事件的方法。这些方法主要用于子类；不能在 AbstractTableModel 中调用。表 19-3 介绍了何时可以调用这些方法。

图 19-7 所示的表格模型是 AbstractTableModel 的一个扩展，该表格显示了 100 行数据，其中每行含有 10 个不变的 Integer 值。

例 19-2 列出了图 19-7 所示应用程序的完整代码。

例 19-2 AbstractTableModel 的一个简单扩展

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.table. * ;
import java.util. * ;

public class Test extends JFrame {
    JTable table = new JTable (
        new AbstractTableModel () {
            int rows = 100, cols = 10;

            public int getRowCount () { return rows; }
            public int getColumnCount () { return cols; }

            public Object getValueAt (int row, int col) {
                return " (" + row + ", " + col + ")";
            }
        }
    );

    public Test () {
        getContentPane ().add (new JScrollPane (table),
            BorderLayout.CENTER);
    }

    public static void main (String args []) {
        GJApp.launch (
            new Test (), "A Simple Model", 300, 300, 450, 300);
    }
}
```

这个应用程序用 AbstractTableModel 的一个内部类扩展来创建一个 JTable 实例。这个模型产生值而不是存储数据。因为该模型的存储要求与 DefaultTableModel 所采用的矢量存储的方式不相符合，所以它扩展 AbstractTableModel 而不是 DefaultTableModel。

类总结 19-1 总结了 AbstractTableModel 类。

类总结 19-1 AbstractTableModel

扩展：java.lang.Object

实现：TableModel、java.io.Serializable

1. 方法

(1) 表格模型监听器

public void addTableModelListener (TableModelListener)


```
public void removeTableModelListener (TableModelListener)
```

用事件监听器列表来维护 TableModelListener 列表。有关 EventListenerList 类的更多信息，请参见“事件监听器列表”一节。

(2) 列

```
public int findColumn (string)
public Class getColumnClass (int)
public String getColumnName (int)
```

AbstractTableModel 按照 A、B、C、...AA、AB、AC... 等样式定义列名。findColumn 方法的返回值是具有特定名字的列的索引，若其名字与列不匹配则返回-1 值。

几乎每个 AbstractTableModel 的扩展都重载 getColumnName () 方法以得到更有意义的列名字。

AbstractTableModel 将它的列定义成一般的 Object 类。AbstractTableModel 的扩展几乎总是重载 getColumnClass ()，以便返回更特殊的类来增加与更合适的绘制器和编辑器匹配的可能性。

(3) 单元和值

```
public boolean isCellEditable (int row, int col)
public void setValueAt (Object value, int row, int col)
```

当 isCellEditable 方法的返回值为 false 时，AbstractTableModel 不允许编辑它的任何单元。

表格模型必须实现由 TableModel 接口定义的 setValueAt () 方法[⊖]。AbstractListModel 提供 setValueAt () 方法的一个空实现以便消除 AbstractListModel 的不变扩展。

(4) 事件激发

```
public void fireTableCellUpdated (int row, int col)
public void fireTableChanged (TableModelEvent)
public void fireTableDataChanged ()
public void fireTableRowsDeleted (int first Row, int last Row)
public void fireTableRowsInserted (int first Row, int last Row)
public void fireTableRowsUpdated (int first Row, int last Row)
public void fireTableStructureChanged ()
```

AbstractTableModel 确定其扩展调用的方法是否完全实现，该方法响应各种模型数据的变化并把事件发送给 TableModelListener 监听器。当修改数据时，扩展 AbstractTableModel 的表格模型必须调用适当的 fire... () 方法。表 19-3 指出每一种方法何时应该调用。

表 19-3 AbstractTableModel 事件激发方法

方法	激发当 ...
fireTableCellUpdated (int row, int column)	单元值已被更新
fireTableDataChanged ()	任何或全部单元值已经改变化
fireTableRowsDeleted (int first, int last)	第一行到最后一行 (包括) 已被删除
fireTableRowsInserted (int first, int last)	第一行到最后一行 (包括) 已被插入
fireTableRowsUpdated (int first, int last)	第一行到最后一行 (包括) 已被更新
fireTableStructureChanged ()	任何表格结构可能已经改变; 等价于设置一个表格的模型

⊖ 为了成为具体 (非抽象) 的类必须这么做。

Swing 提示

AbstractTableModel 扩展必须激发相应的事件

AbstractTableModel 提供七种方法，这些方法被用于在各种数据修改案例下激发 TableModelEvent 事件。例如，如果删除第三行至第七行，那么在删除后，AbstractTableModel 的一个扩展将调用方法 AbstractTableModel.fireTableRowsDeleted (3, 7)。同样情况下，如果修改单个单元值，那么将调用方法 AbstractTableModel.fireTableCellUpdated ()。

表示可变数据的表格模型必须实现方法 setValueAt ()。因为 setValueAt () 修改一个表格单元的值，所以该方法必须激发一个表格单元更新事件，这可以由 AbstractTableModel.fireTableCellUpdated () 方法完成。

19.2.4 DefaultTableModel

DefaultTableModel 扩展 AbstractTableModel 并实现数据操纵方法，这些方法是其超类未实现的。DefaultTableModel 将单元值存储在代表行的矢量中，该矢量中的每个对象也是一个对象矢量，这些对象代表单元值。例如，可以用下面的方式来访问行 1 列 2 的单元值：

```
DefaultTableModel defaultTableModel = ...
Vector dataVector = defaultTableModel.getDataVector ();
Vector rowVector = (Vector) dataVector.elementAt (1);
Object cellValue = (Vector) rowVector.elementAt (2);
```

除提供一个存储机制之外，DefaultTableModel 还提供添加列和添加、插入和删除行的方法。可以用 DefaultTableModel 的 setValueAt () 实现来修改单独的单元值，setValueAt () 存储指定的值并激发一个 TableModelEvent 事件。

使用 DefaultTableModel 的主要动机就是它在构造模型后能够添加和删除数据。

DefaultTableModel 的存储机制不太重要，因为大多数时候必须把数据重新格式化为模型的一组矢量。对有些（不可否认有很多）表格，这种数据重新格式化的代价是昂贵的，而将数据以原来格式保存对扩展 AbstractTableModel 和实现那三种数据访问方法所付出的代价要少得多。另一方面，通过继承并在 DefaultTableModel 帮助下实现添加、插入或删除列的方法可能是一次更具有挑战的经历。

如图 19-8 所示的应用程序包含一个表格，这个表格配备了 DefaultTableModel 的一个实例。该应用程序提供添加行和列的按钮，并显示了 DefaultTableModel 修改数据的能力。

图 19-8 中的上图显示了这个应用程序的初始状况。下图显示了添加三行和三列

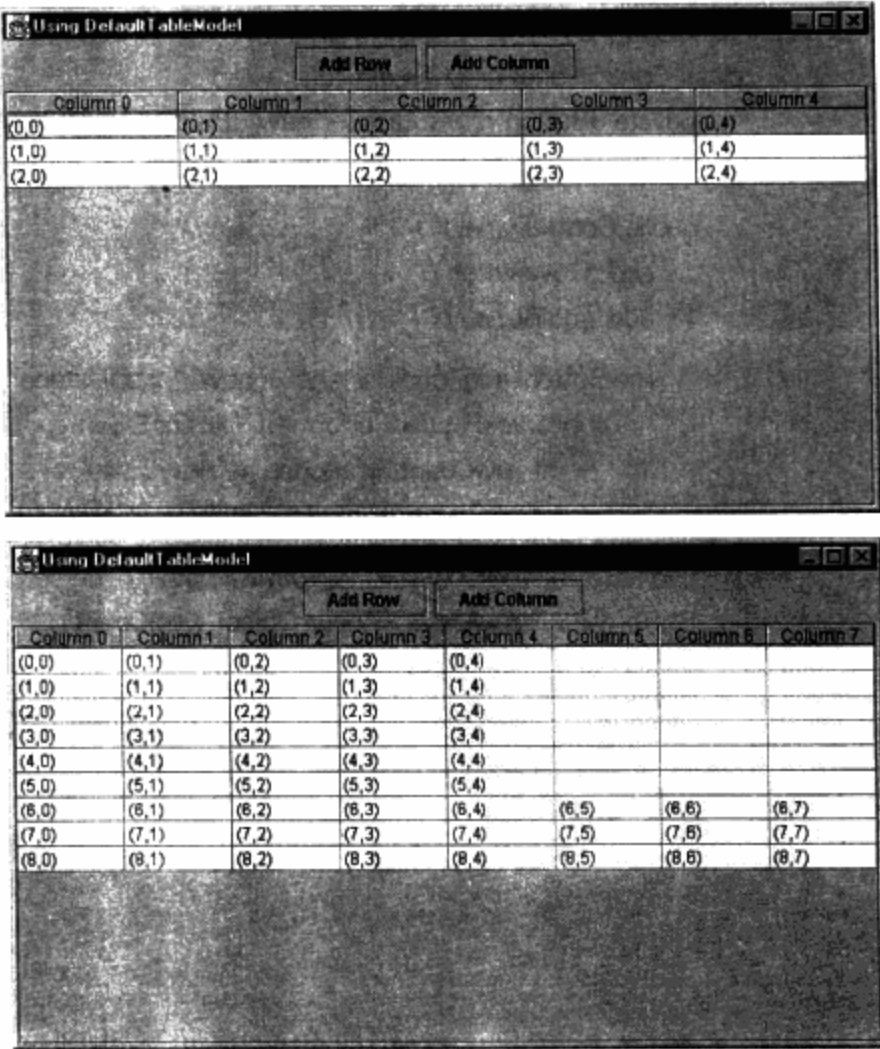


图 19-8 使用 DefaultTableModel

后这个应用程序的样子。

表格模型初始化为三行和五列。首先利用 `DefaultTableModel.addColumn()` 方法把列添加到模型中，然后用 `DefaultTableModel.addRow()` 来添加行。

```
public class Test extends JFrame {
    private int rows = 3, cols = 5;
    private Object [] rowData = new Object [cols];

    private DefaultTableModel model = new DefaultTableModel ();
    private JTable table = new JTable (model);

    public Test () {
        for (int c=0; c < cols; ++c)
            model.addColumn ("Column " + Integer.toString (c));

        for (int r=0; r < rows; ++r) {
            for (int c=0; c < cols; ++c) {
                rowData [c] = " (" + r + ", " + c + ")";
            }
            model.addRow (rowData);
        }

        getContentPane ().add (new JScrollPane (table),
                                BorderLayout.CENTER);
        getContentPane ().add (new ControlPanel (),
                                BorderLayout.NORTH);
    }
}
```

这个应用程序的按钮有动作监听器，它把数据添加到模型中。如果已添加了列，Add Row 按钮将分配 `rowData` 数组并把已命名的字符串插入到该数组，随后添加这些数据到模型中。

```
...
class ControlPanel extends JPanel {
    private JButton rowButton = new JButton("Add (Row)",
        colButton = new JButton ("Add Column");

    public ControlPanel () {
        add (rowButton);
        add (colButton);

        rowButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int rowCount = model.getRowCount ();
                int colCount = model.getColumnCount ();

                if (colCount > rowData.length)
                    rowData = new Object [colCount];

                for (int c=0; c < colCount; ++c) {
                    rowData [c] = " (" + rowCount + ", " +
                        c + ")";
                }
                model.addRow (rowData);
            }
        });
    }
}
```

Add Column 按钮把一个相应命名的列添加到这个模型中。必须调用 `JTable.sizeColumnsToFit()` 方法，这是因为 `JTable` 有一个错误，即在已添加或删除列后不能正确地更新表格。

```

        ...
        colButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int colCount = model.getColumnCount ();
                model.addColumn ("Column " + colCount);

                // Bug: the call to sizeColumnsToFit ()
                // should not be necessary
                table.sizeColumnsToFit (-1);
            }
        });
    }
}

```

例 19-3 列出了图 19-3 所示应用程序的完整代码。

例 19-3 使用 DefaultTableModel

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.table. * ;
import java.util. * ;

public class Test extends JFrame {
    private int rows = 3, cols = 5;
    private Object [ ] rowData = new Object [cols];

    private DefaultTableModel model = new DefaultTableModel ();
    private JTable table = new JTable (model);

    public Test () {
        for (int c = 0; c < cols; ++c)
            model.addColumn ("Column " + Integer.toString (c));

        for (int r = 0; r < rows; ++r) {
            for (int c = 0; c < cols; ++c) {
                rowData [c] = " (" + r + ", " + c + ")";
            }
            model.addRow (rowData);
        }

        getContentPane ().add (new JScrollPane (table),
                                BorderLayout.CENTER);
        getContentPane ().add (new ControlPanel (),
                                BorderLayout.NORTH);
    }

    public static void main (String args []) {
        GJApp.launch (new Test (),
                      "Using DefaultTableModel", 150, 150, 600, 350);
    }

    class ControlPanel extends JPanel {
        private JButton rowButton = new JButton ("Add Row"),
                      colButton = new JButton ("Add Column");

        public ControlPanel () {
            add (rowButton);
            add (colButton);

            rowButton.addActionListener (new ActionListener () {

```

```

        public void actionPerformed (ActionEvent e) {
            int rowCount = model.getRowCount ();
            int colCount = model.getColumnCount ();

            if (colCount > rowData.length)
                rowData = new Object [colCount];

            for (int c=0; c < colCount; ++c) {
                rowData [c] = " (" + rowCount + "," +
                    c + ")";
            }

            model.addRow (rowData);
        }

        colButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int colCount = model.getColumnCount ();
                model.addColumn ("Column " + colCount);

                // Bug: the call to sizeColumnsToFit ()
                // should not be necessary
                table.sizeColumnsToFit (-1);
            }
        });
    }
}

```

类总结 19-2 总结了 DefaultTableModel 类。

类总结 19-2 DefaultTableModel

扩展: AbstractTableModel

实现: java.io.Serializable

1. 构造方法

```

public DefaultTableModel ()
public DefaultTableModel (int numRows, int numColumns)
public DefaultTableModel (Object [] columnNames, int numRows)
public DefaultTableModel (Object [][] data, Object [] columnNames)
public DefaultTableModel (Vector columnNames, int numRows)
public DefaultTableModel (Vector data, Vector columnNames)

```

DefaultTableModel 提供构造方法, 这些构造方法以一个或多个值作为参数, 这些值是: 行数或列数、列名字和表格数据。此外, 还提供了—个无参数的构造方法。

2. 方法

```

protected static Vector convertToVector (Object [])
protected static Vector convertToVector (Object [][])

```

上面所列的方便方法用于把列名字和表格数据转化成矢量。这些方法是 protected, 因此, 可以使用 DefaultTableModel 的扩展。

3. 数据

```

public Vector getDataVector ()
public void setDataVector (Object [][] data, Object [] columnNames)

```

```

public void setDataVector (Vector data, Vector columnNames)
public void newDataAvailable (TableModelEvent)
public Object getValueAt (int row, int column)
public void setValueAt (Object value, int row, int column)
public boolean isCellEditable (int row, int column)

```

DefaultTableModel 为其数据矢量提供设置和获取的访问方法。newDataAvailable()方法与 AbstractTableModel.fireTableChanged()方法相同,不能在 DefaultTableModel 中调用 newDataAvailable()方法,也不能在 Swing 中的任何地方调用^①newDataAvailable()方法。

上面所列的最后三种方法由 TableModel 接口来定义。DefaultTableModel 定义所有单元都是可编辑的,与它的超类 (AbstractTableModel) 相反,AbstractTableModel 定义所有单元都是不可编辑的。DefaultTableModel.setValueAt()激发一个 TableModelEvent 事件,指出一个单元值已更新。

4. 列

```

public void addColumn (Object columnIdentifier)
public void addColumn (Object columnIdentifier, Object [] columnData)
public void addColumn (Object columnIdentifier, Vector columnData)

public void setColumnIdentifiers (Object [] columnIdentifiers)
public void setColumnIdentifiers (Vector columnIdentifiers)

public int getColumnCount ()
public String getColumnName (int index)

```

DefaultTableModel 提供添加列的方法纯粹是为了操作方便,还可将表格列添加到一个表格的列模型中。如果需要移动或删除列,那么必须使用一个表格列模型。

传送给上面所列的前五种方法的标识符被显示在列的头部。标识符通常是一些字符串。

getColumnName 方法返回 ColumnIdentifier.toString(),其中 columnIdentifier 是询问中的列的一个相应标识符。

5. 行

```

public void addRow (Object [] rowData)
public void addRow (Vector rowData)
public int getRowCount ()

public void insertRow (int index, Object [] rowData)
public void insertRow (int index, Vector rowData)
public void moveRow (int startIndex, int endIndex, int toIndex)
public void newRowsAdded (TableModelEvent)
public void removeRow (int index)
public void rowsRemoved (TableModelEvent)

public void setNumRows (int numRows)

```

虽然列是对象,但是行是一种由表格模型定义的逻辑结构。通常情况下,列不是通过 DefaultTableModel 的实例而是用 TableColumnModel 和 TableColumn 类来进行操纵的。另外,因为行不是对象 (例如,在行中不存在 TableRowModel 或 TableRow 类),所以 DefaultTableModel 必须提供足够的面向行的方法以满足开发人员的需要。

Swing 提示

① 方法 newDataAvailable 可能在 Swing 以后的版本中消失。

DefaultTableModel 的 Pros 和 Cons

DefaultTableModel 除了具有其超类 (AbstractTableModel) 提供的基本功能外还有两个主要特性：数据存储和可变性。

数据存储是以一组矢量的方式来实现的，如果数据在构造期间以不同形式被定义，那么把该数据的引用复制到模型的矢量中。有这样一种缺省的存储机制是非常方便的，然而在执行和设计的考虑中经常需要花很大努力来实现定制的存储机制。在运行期间修改表格数据并不总是需要的，但是具有这样的功能总是好的。遗憾的是，数据的修改不能与数据存储的方式分开，因此 DefaultTableModel 将数据的可变性维系在矢量的存储机制上。

19.2.5 表格模型、缺省绘制器和缺省编辑器

JTable 类维护一组缺省的绘制器和缺省的编辑器，它们用于连结从 getColumnClass 方法返回的类，getColumnClass 方法在 TableModel 接口中定义。例如，如果没有为一个表格列显式地指定一个绘制器，并且表格模型的 getColumnClass 方法返回 Boolean 类，那么将使用一个复选框来绘制列中的值。同样，如果为列显式地指定一个编辑器，那么也用一个复选框来编辑 Boolean 值。

表 19-4 列举了具有缺省绘制器和缺省编辑器^①的类和用于绘制和编辑表格单元值的组件。

表 19-4 类的缺省绘制器和缺省编辑器

类	绘制器组件	编辑器组件
Object	JLabel	JTextField
Date	JLabel (右排列) ^①	JTextField
Number	JLabel (右排列) ^②	JTextField (右排列)
ImageIcon	JLabel (中央排列)	——
Boolean	JCheckBox (中央排列)	JCheckBox (中央排列)

① 绘制器使用 java.text.NumberFormat 来格式化数字。
② 绘制器使用 java.text.DateFormat 来格式化日期。

图 19-9 显示的表格是一个用于旅馆房间预订的应用程序。用一个对象数组来指定这个表格的行数据：名字用 string，入住和退房日期用 java.util.Date；吸烟、传真和便携电脑用 boolean，房间使用率用 Double，照片用 ImageIcon。

这个应用程序创建一个代表列名的字符串数组，并创建代表单元值的一个二维 Object 数组。这些数组都被传送给 DefaultTableModel 的构造方法。

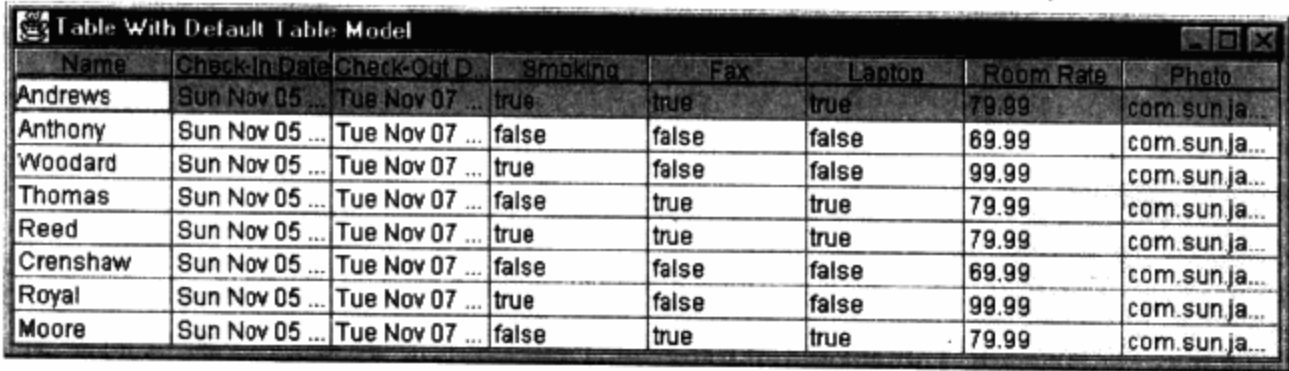


图 19-9 使用 DefaultTableModel

```
public class Test extends JFrame {
    String [] columnNames = {
        "Name", "Check-In Date", "Check-Out Date", "Smoking",
        "Fax", "Laptop", "Room Rate", "Photo",
    }
```

① 对 Swing 1.1FCS 而言。


```
};

Date dayOne = (new GregorianCalendar (2000, 10, 5)) .getTime ();
Date dayTwo = (new GregorianCalendar (2000, 10, 7)) .getTime ();

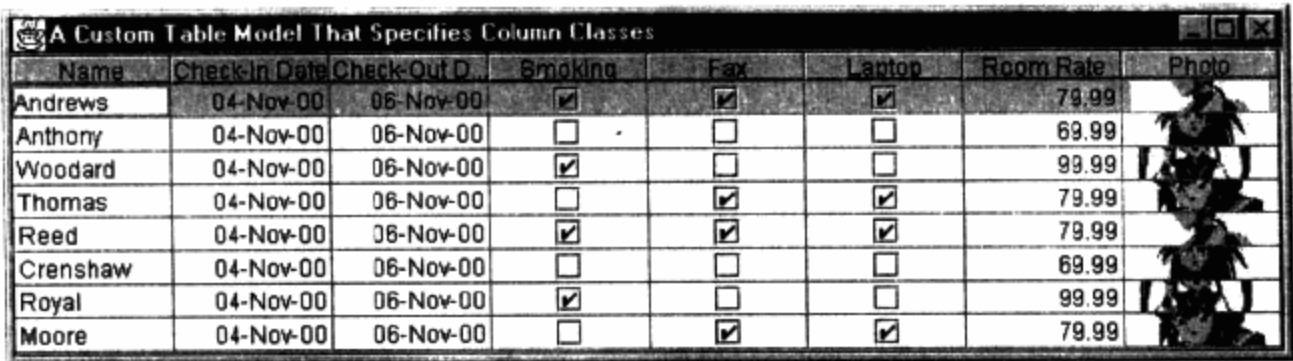
Object [] [] data = {
    { "Andrews", dayOne, dayTwo,
      new Boolean (true), new Boolean (true),
      new Boolean (true), new Double (79.99),
      new ImageIcon ("tenchi.jpg") },
    { "Anthony", dayOne, dayTwo,
      new Boolean (false), new Boolean (false),
      new Boolean (false), new Double (69.99),
      new ImageIcon ("washu.jpg") },
    ...
};

JTable table = new JTable (data, columnNames);

public Test () {
    getContentPane ().add (new JScrollPane (table),
                             BorderLayout.CENTER);
}

...
}
```

显然，图 19-9 显示的表格需要改进。图 19-10 显示的是同一张表格，这次使用缺省绘制器来绘制 Date，Boolean 和 Double 值。使用缺省绘制器后明显改善了图 19-9 所示的表格，然而绘制和编辑仍有许多需要改进的地方，这些问题将在本章讨论。







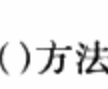



Name	Check-In Date	Check-Out Date	Smoking	Fax	Laptop	Room Rate	Photo
Andrews	04-Nov-00	06-Nov-00	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	79.99	
Anthony	04-Nov-00	06-Nov-00	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	69.99	
Woodard	04-Nov-00	06-Nov-00	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	99.99	
Thomas	04-Nov-00	06-Nov-00	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	79.99	
Reed	04-Nov-00	06-Nov-00	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	79.99	
Crenshaw	04-Nov-00	06-Nov-00	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	69.99	
Royal	04-Nov-00	06-Nov-00	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	99.99	
Moore	04-Nov-00	06-Nov-00	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	79.99	

图 19-10 缺省绘制器和编辑器

这个表格的模型是 DefaultTableModel 的一个扩展，它实现 getColumnClass () 方法以便返回相应列的首行值的类。

```
class CustomModel extends DefaultTableModel {
    public CustomModel (Object [] [] data, Object [] columnNames) {
        super (data, columnNames);
    }

    public Class getColumnClass (int col) {
        // dataVector is a protected member of DefaultTableModel

        Vector v = (Vector) dataVector.elementAt (0);
        return v.elementAt (col) .getClass ();
    }
}

...
}
```

虽然从图 19-10 中看得不太明显，但使用缺省编辑器，表格中 Boolean 和 Double 值都是可编辑的。通过重载表格模型中的方法 isCellEditable ()，可以使编辑有效。如果列的类是除 Im-

ageIcon 或 java.util.Date 以外的任何类，那么标识这个单元为是可编辑的。

```
public boolean isCellEditable (int row, int col) {
    Class columnClass = getColumnClass (col);
    return columnClass != ImageIcon.class &&
           columnClass != Date.class;
}
```

例 19-4 列出了图 19-10 所示应用程序的完整代码。

例 19-4 使用缺省绘制器和编辑器

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;

public class Test extends JFrame {
    String [] columnNames = {
        "Name", "Check-In Date", "Check-Out Date", "Smoking",
        "Fax", "Laptop", "Room Rate", "Photo",
    };

    Date dayOne = (new GregorianCalendar (2000, 10, 5)).getTime ();
    Date dayTwo = (new GregorianCalendar (2000, 10, 7)).getTime ();
    Object [] [] data = {
        //lengthy listing omitted for brevity
    };

    JTable table = new JTable (new CustomModel (data, columnNames));

    public Test () {
        getContentPane ().add (new JScrollPane (table),
            BorderLayout.CENTER);
    }

    public static void main (String args []) {
        GJApp.launch (
            new Test (),
            "A Custom Table Model That Specifies Column Classes",
            300, 300, 650, 182);
    }
}

class CustomModel extends DefaultTableModel {
    public CustomModel (Object [] [] data, Object [] columnNames) {
        super (data, columnNames);
    }

    public Class getColumnClass (int col) {
        // dataVector is a protected member of DefaultTableModel

        Vector v = (Vector) dataVector.elementAt (0);
        return v.elementAt (col).getClass ();
    }

    public boolean isCellEditable (int row, int col) {
        Class columnClass = getColumnClass (col);
        return columnClass != ImageIcon.class &&
           columnClass != Date.class;
    }
}
```

Swing 提示

影响绘制和编辑的两个 TableModel 方法

使用以下的 TableModel 方法来适应具有缺省绘制器和编辑器的表格列：

```
public Class getColumnClass ()
public boolean isCellEditable ()
```

如果从 TableModel.getColumnClass()方法返回的类是预确定的类集中的一个类的话，则指定列中的值配备了缺省绘制器和（或）编辑器，这个类集由 JTable 类来定义，参见“表格模型、缺省绘制器和编辑器”一节。实际中是否使用编辑器取决于从表格模型的 isCellEditable 方法返回的值。如果这个方法返回 false 值，那么这个单元值不能被编辑。

19.3 表格列

Swing 表格是面向列的，例如，表格数据是基于每一列来进行绘制和编辑的。面向列的进一步证据就是，table 包为表格各列提供类，但表格的各行却没有。因此，表格列用 TableColumn 类来表示，它是 Swing 表格的支柱。

表格列用 TableColumn 类来表示。表格列不是组件，TableColumn 类只简单地维护表格列的属性。

TableColumn 类唯一激发的事件是属性变化事件，该事件在其关联属性修改时被激发。图 19-11 显示了 TableColumn 类。

TableColumn 的实例有两个绘制器——一个用于列头部而另一个用于它的单元。表格列还有一个单元值编辑器。

表格列显示了一个作为列头部值引用的对象。头部值是 TableColumn 类的属性并由列头部绘制器进行绘制。

表格列不控制表格的高度，表格的高度由表格的行数和行高决定，但表格列控制表格的宽度。事实上，TableColumn 的实例跟踪表格列实际的、首选的宽度以及最小和最大宽度。缺省时，表格列是可以调整大小的，但它们的宽度在构造表格列后可以被固定。

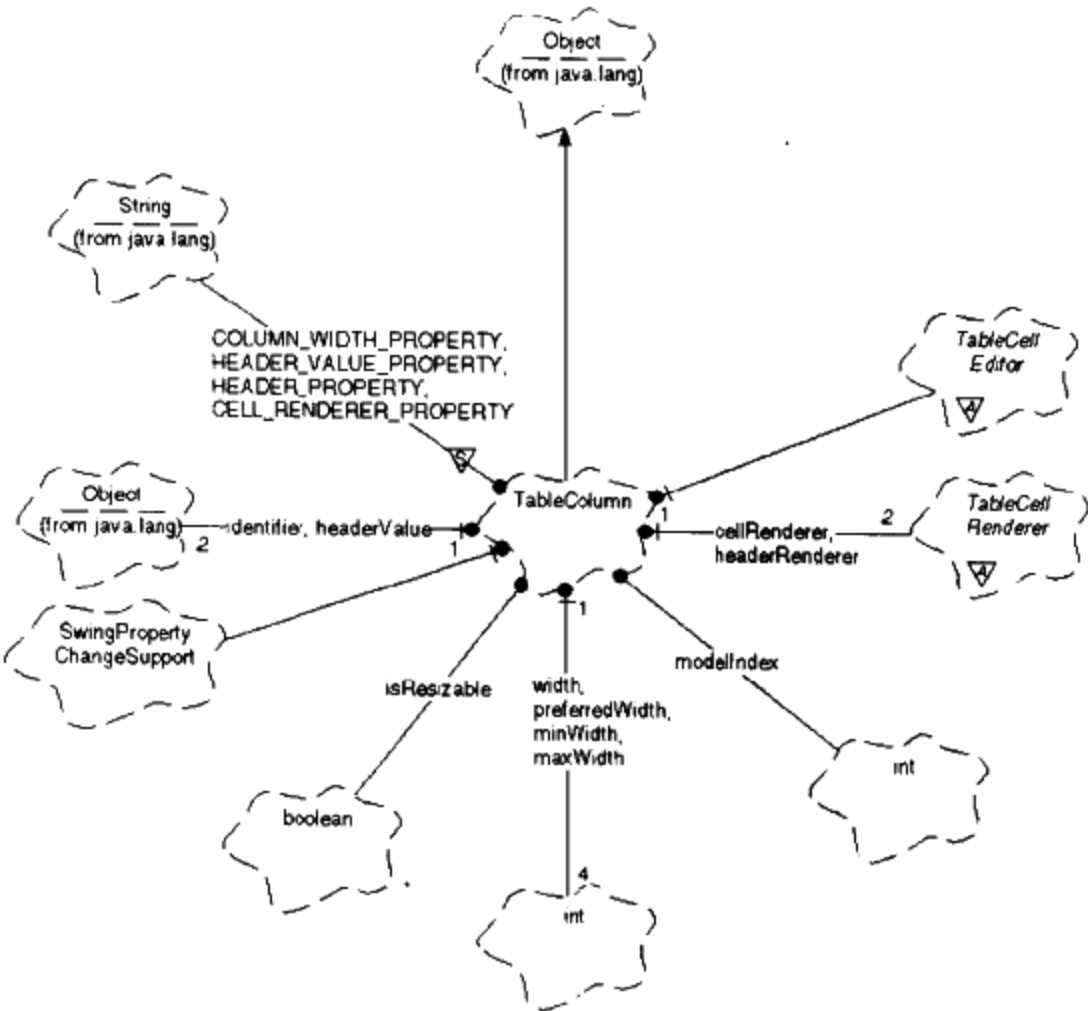


图 19-11 TableColumn 类图

表格列保留了 Swing 表格早期设计的一部分特点。表格各列由 Object 标识符进行识别。后来，JTable 采用基于整体列的独特访问方式，Swing 类不再使用列标识符。但是，TableColumn 类保留了它的标识符属性和相应的访问方法。列标识符，若未明确设置，则缺省认为是列的头部值。

列可以在表格中排序，但显示在列中的数据仍保存在模型中。换言之，即使列在屏幕上排序，存储在表格模型中的数据也不会因为排序而移动。维护两种列顺序——一是视屏上，另一种是在模型中，由在每列中保存的模型索引来完成。

在 SwingPropertyChangeSupport 对象的协助下，当 TableColumn 的关联属性被修改时，它激发属性改变事件。TableColumn 还提供 public 字符串，属性变化监听器用这些字符串来标识修改了哪个属性。

19.3.1 列调整大小模式

当调整列的大小时，该列得到或失去的空间必须是来自或让与这个表格其他的列。JTable 类提供了五种列调整大小模式，它们以表 19-5 所描述的方式释放空间。

表 19-5 JTable 列调整大小模式

模式	空间来自/让与 ...
AUTO_RESIZE_OFF	正在调整大小的列
AUTO_RESIZE_LAST_COLUMN	最右边的列
AUTO_RESIZE_SUBSEQUENT_COLUMNS	正在调整大小的列右边的所有列
AUTO_RESIZE_NEXT_COLUMN	正在调整大小的列的右边一列
AUTO_RESIZE_ALL_COLUMNS	所有列

图 19-12 显示了允许设置表格调整大小模式的一个应用程序。这个应用程序说明了以何种方式来设置表格调整大小模式，并提供了一个 JTable 调整大小模式的可视描述。

图 19-12 所示的应用程序创建了两个数组，一个含有表示表格调整大小模式的字符串，另一个含有相应的整型常量。这个应用程序还创建了一个六行五列的 JTable 实例。表格被放入一个滚动窗格中，并作为中央组件被添加到该应用程序的内容窗格中。

该组合框包含在 JPanel 一个内部类的扩展中，并作为上边组件添加到这个应用程序的内容窗格中。

```
public class Test extends JFrame {
    Object [] resizeModees = new Object [] {
        "JTable.AUTO_RESIZE_OFF",
        "JTable.AUTO_RESIZE_NEXT_COLUMN",
        "JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS",
        "JTable.AUTO_RESIZE_LAST_COLUMN",
        "JTable.AUTO_RESIZE_ALL_COLUMNS",
    };
    int [] resizeConstants = {
        JTable.AUTO_RESIZE_OFF,
        JTable.AUTO_RESIZE_NEXT_COLUMN,
        JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS,
        JTable.AUTO_RESIZE_LAST_COLUMN,
        JTable.AUTO_RESIZE_ALL_COLUMNS,
    };
    JTable table = new JTable (6, 5);
```

```
public Test () {  
    Container contentPane = getContentPane ();  
    contentPane.add (new ControlPanel (), BorderLayout.NORTH);  
    contentPane.add (new JScrollPane (table),  
        BorderLayout.CENTER);  
}
```

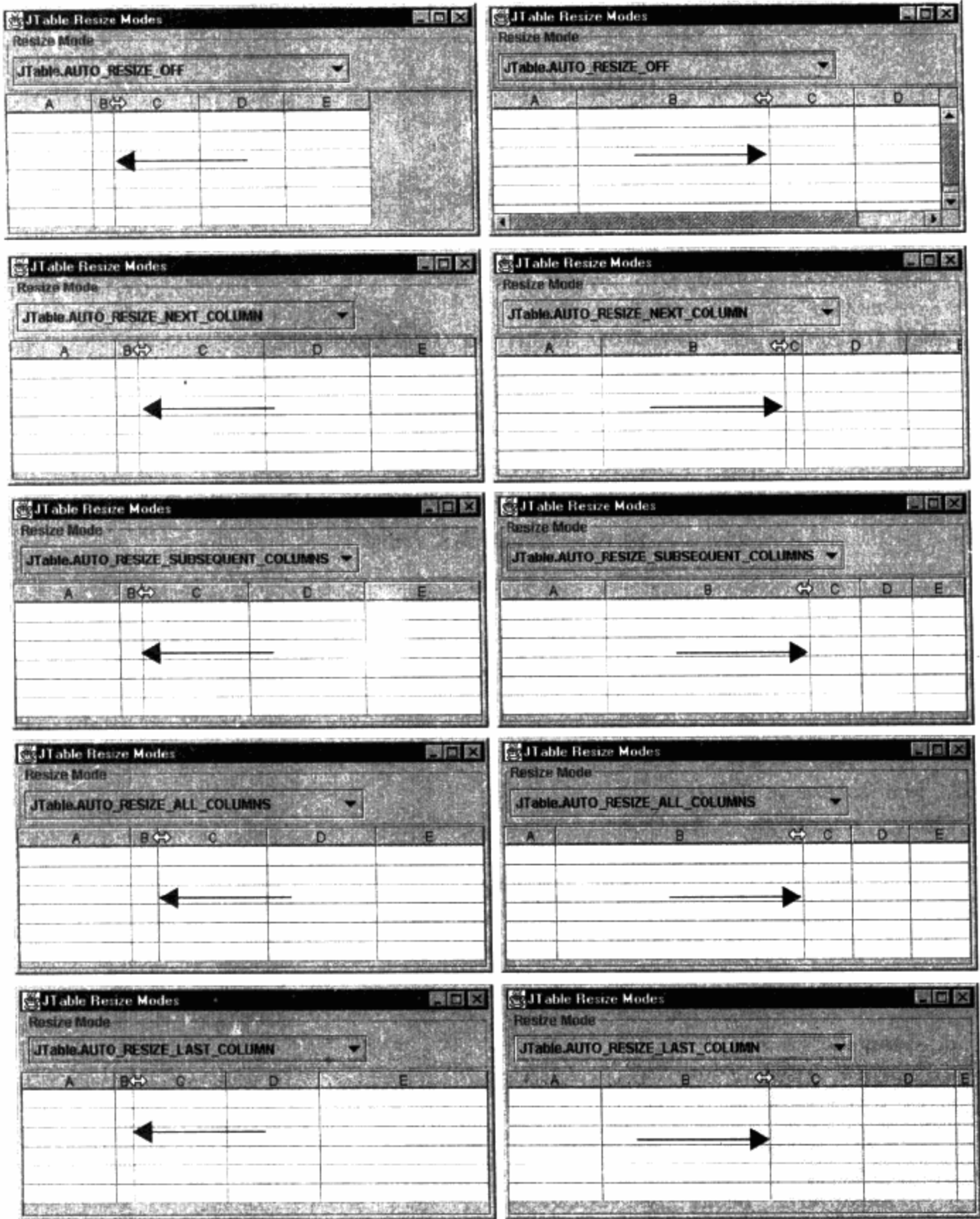


图 19-12 JTable 调整大小模式

该组合框用字符串数组来构造并由获得的表格的初始调整大小模式来初始化。
添加到组合框中的监听器根据从组合框中选取的选项来设置表格调整大小模式。

```
...  
class ControlPanel extends JPanel {  
    JComboBox resizeModeCombo = new JComboBox (resizeModes);
```

```

public ControlPanel () {
    initializeCombo ();
    ...
    resizeModeCombo.addActionListener (
        new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                int index =
                    resizeModeCombo.getSelectedIndex ();
                table.setAutoResizeMode (
                    resizeConstants [index]);
            }
        });
}

private void initializeCombo () {
    int resizeMode = table.getAutoResizeMode ();
    if (resizeMode == JTable.AUTO_RESIZE_OFF)
        resizeModeCombo.setSelectedIndex (0);
    else if (resizeMode == JTable.AUTO_RESIZE_NEXT_COLUMN)
        resizeModeCombo.setSelectedIndex (1);
    ...
}
}

```

例 19-5 列出了图 19-12 所示应用程序的完整代码。

例 19-5 JTable 调整大小模式

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    Object [] resizeModees = new Object [] {
        "JTable.AUTO_RESIZE_OFF",
        "JTable.AUTO_RESIZE_NEXT_COLUMN",
        "JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS",
        "JTable.AUTO_RESIZE_LAST_COLUMN",
        "JTable.AUTO_RESIZE_ALL_COLUMNS",
    };

    int [] resizeConstants = {
        JTable.AUTO_RESIZE_OFF,
        JTable.AUTO_RESIZE_NEXT_COLUMN,
        JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS,
        JTable.AUTO_RESIZE_LAST_COLUMN,
        JTable.AUTO_RESIZE_ALL_COLUMNS,
    };

    JTable table = new JTable (6, 5);

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.add (new ControlPanel (), BorderLayout.NORTH);
        contentPane.add (new JScrollPane (table),
            BorderLayout.CENTER);
    }
}

```

```

class ControlPanel extends JPanel {
    JComboBox resizeModeCombo = new JComboBox (resizeModes);

    public ControlPanel () {
        initializeCombo ();

        setBorder (BorderFactory.createTitledBorder (
            "Resize Mode"));

        setLayout (new FlowLayout (FlowLayout.LEFT, 2, 2));
        add (resizeModeCombo);

        resizeModeCombo.addActionListener (
            new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    int index =
                        resizeModeCombo.getSelectedIndex ();

                    table.setAutoResizeMode (
                        resizeConstants [index]);
                }
            }
        );
    }

    private void initializeCombo () {
        int resizeMode = table.getAutoResizeMode ();

        if (resizeMode == JTable.AUTO_RESIZE_OFF)
            resizeModeCombo.setSelectedIndex (0);
        else if (resizeMode == JTable.AUTO_RESIZE_NEXT_COLUMN)
            resizeModeCombo.setSelectedIndex (1);
        else if (resizeMode == JTable.AUTO_RESIZE_LAST_COLUMN)
            resizeModeCombo.setSelectedIndex (2);
        else if (resizeMode == JTable.AUTO_RESIZE_ALL_COLUMNS)
            resizeModeCombo.setSelectedIndex (3);
        else if (
            resizeMode == JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS)
            resizeModeCombo.setSelectedIndex (4);
    }

    public static void main (String args []) {
        GJApp.launch (
            new Test (). "JTable Resize Modes", 300, 300, 425, 210);
    }
}

```

19.3.2 列宽度

通常情况，表格列并未设置成缺省的宽度。例如，图 19-13 上图的应用程序含有一个表格，它的三个列具有相同的宽度。MI (middle initial) 列比它所需要的大小宽得多，而 LastName 列的宽度又不足以容纳其最宽的项。

图 19-13 下图所示的应用程序修改了它的列宽，MI 列仅占用它所需要的宽度，LastName 列的宽足够容纳它的最大的项。

如图 19-13 所示，列调整大小所需要的一切就是一个用于计算列的 actual 首选宽度的方法。尽管列维护由方法 `TableColumn.getPreferredWidth ()` 访问的 `preferredWidth` 属性，列仍有一个缺

省为 75 个像素的首选宽度。换言之，一个列的首选宽度在缺省情况下不是一个计算值。

一个列的 actual 首选宽度足以容纳列头部和单元内被绘制的组件。图 19-13 下图所示的应用程序实现一个方法来计算一个列的实际的首选宽度。

图 19-13 下图所示的应用程序创建一个具有列名字数组和表示 first name、last name 和 middle initial 二维字符串数组的表格。

```
public class Test extends JFrame {
    Object [] columnNames =
        {"First Name", "MI", "Last Name"};

    Object [][] names = {
        {"Lynn", "M.", "Seckinger"},
        {"Carol", "R.", "Seckinger"},
        {"Roy", "D.", "Martin"},
        {"Bill", "O.", "Veryveryveryverylonglastname"},
        {"Richard", "A.", "Tattersall"},
        {"Philip", "B.", "Edwards"},
        {"Moore", "T.", "Moore"}
    };

    JTable table = new JTable (names, columnNames);
    ...
}
```

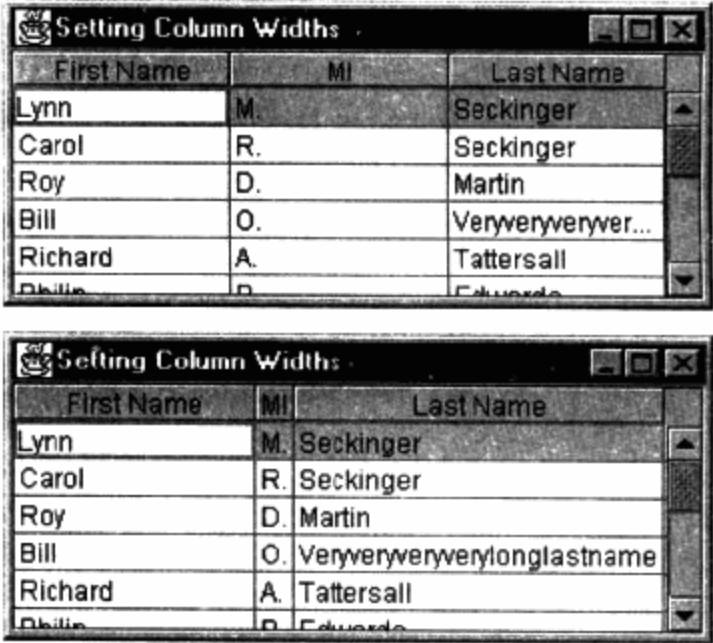


图 19-13 指定列宽度

考虑到列头部和单元值绘制的方式，该应用程序的构造方法调用方法 `getPreferredWidthForColumn`（下面列出）来返回列的实际首选宽度。

MI 列用它的最小和最大宽度值设置首选宽度，即以首选宽度作为列的宽度。把 Last Name 列的最小宽度值设置成该列的首选宽度，这样该列的宽度不会比其首选宽度更小。

```
...
public Test () {
    TableColumn mid = table.getColumnModel ().getColumn (columnNames [1]);
    TableColumn last = table.getColumnModel ().getColumn (columnNames [2]);

    int midWidth = getPreferredWidthForColumn (mid),
        lastWidth = getPreferredWidthForColumn (last);

    mid.setMinWidth (midWidth);
    mid.setMaxWidth (midWidth);
    last.setMinWidth (lastWidth);

    // sizeColumnsToFit () must be called due to a JTable
    // bug ...
    table.sizeColumnsToFit (0);
    ...
}
...
```

注意 由于一个 JTable 错误，`JTable.sizeColumnToFit ()` 必须被调用。

下面所列的方法 `getPreferredWidthForColumn ()` 计算列的实际首选宽度值。通过调用这个应用程序的两个 private 方法来获得列头部和单元的实际首选宽度。从 `getPreferredWidthForColumn` 方法中返回这两个值中较宽的值。

```
...
public int getPreferredWidthForColumn (TableColumn col) {
    int hw = columnHeaderWidth (col), // hw = header width
        cw = widestCellInColumn (col); // cw = column width
```

```
return hw > cw ? hw : cw;
```

`columnHeaderWidth` 方法计算列头部的实际首选宽度。首先, 利用 `TableColumn.getHeaderRenderer` 方法获得列头部绘制器的一个引用。然后, 利用这个绘制器来获得对这个绘制器组件的一个引用。最后得到这个组件的首选大小。有关表格头部和单元绘制器的更多信息, 请参见 19.9 节“表格头部”。

列头部绘制器实现 `TableCellRenderer` 接口, 该接口定义了方法 `getTableCellRendererComponent`, 并将行和列作为该方法的参数。对于头部绘制器 (不像单元绘制器), 传送给 `getTableCellRendererComponent` 的行列值被忽略, 因而 `columnHeaderWidth` 方法将 0 作为行列值传送给 `getTableCellRendererComponent` 方法。

```
...
private int columnHeaderWidth (TableColumn col) {
    TableCellRenderer renderer = col.getHeaderRenderer ();
    Component comp = renderer.getTableCellRendererComponent (
        table, col.getHeaderValue (),
        false, false, 0, 0);
    return comp.getPreferredSize ().width;
}
...
```

`widestCellInColumn` 方法获得对列单元绘制器和这个绘制器组件的一个引用。该方法的返回值是组件首选宽度的最大值。

```
...
private int widestCellInColumn (TableColumn col) {
    int c = col.getModelIndex (), width = 0, maxw = 0;
    for (int r = 0; r < table.getRowCount (); ++r) {
        TableCellRenderer renderer =
            table.getCellRenderer (r, c);
        Component comp =
            renderer.getTableCellRendererComponent (
                table, table.getValueAt (r, c),
                false, false, r, c);
        width = comp.getPreferredSize ().width;
        maxw = width > maxw ? width : maxw;
    }
    return maxw;
}
...
```

应该注意到 `TableColumn` 类提供了一个 `getCellRenderer()` 方法, 这个方法可以由上面列出的方法使用而 `JTable.getCellRenderer (int row, int col)` 不能。但使用 `JTable` 方法具有这样的优点, 如果列的绘制器未明确设置, 那么 `JTable.getCellRenderer()` 将返回缺省的绘制器给列数据所在的类。

例 19-6 列出了图 19-13 下图所示应用程序的完整代码。

例 19-6 指定列宽度

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```

import javax.swing.table.*;
import java.util.*;

public class Test extends JFrame {
    Object [] columnNames =
        {"First Name", "MI", "Last Name"};

    Object [][] names = {
        {"Lynn", "M.", "Seckinger"},
        {"Carol", "R.", "Seckinger"},
        {"Roy", "D.", "Martin"},
        {"Bill", "O.", "Veryveryveryverylonglastname"},
        {"Richard", "A.", "Tattersall"},
        {"Philip", "B.", "Edwards"},
        {"Moore", "T.", "Moore"}
    };

    // shorten scrollbar grip with these ...
    {"Lynn", "M.", "Seckinger"},
    {"Carol", "R.", "Seckinger"},
    {"Roy", "D.", "Martin"},
    {"Bill", "O.", "Veryveryveryverylonglastname"},
    {"Richard", "A.", "Tattersall"},
    {"Philip", "B.", "Edwards"},
    {"Moore", "T.", "Moore"};

    JTable table = new JTable (names, columnNames);

    public Test () {
        TableColumn mid = table.getColumnModel ().getColumn (columnNames [1]);
        TableColumn last = table.getColumnModel ().getColumn (columnNames [2]);

        int midWidth = getPreferredWidthForColumn (mid),
            lastWidth = getPreferredWidthForColumn (last);

        mid.setMinWidth (midWidth);
        mid.setMaxWidth (midWidth);
        last.setMinWidth (lastWidth);

        getContentPane ().add (new JScrollPane (table),
                                BorderLayout.CENTER);
    }

    public int getPreferredWidthForColumn (TableColumn col) {
        int hw = columnHeaderWidth (col), // hw = header width
            cw = widestCellInColumn (col); // cw = column width

        return hw > cw ? hw : cw;
    }

    private int columnHeaderWidth (TableColumn col) {
        TableCellRenderer renderer = col.getHeaderRenderer ();
        Component comp = renderer.getTableCellRendererComponent (
            table, col.getHeaderValue (),
            false, false, 0, 0);

        return comp.getPreferredSize ().width;
    }

    private int widestCellInColumn (TableColumn col) {
        int c = col.getModelIndex (), width = 0, maxw = 0;

        for (int r = 0; r < table.getRowCount (); ++r) {
            TableCellRenderer renderer =

```

```

        table.getCellRenderer (r, c);
        Component comp =
            renderer.getTableCellRendererComponent (
                table, table.getValueAt (r, c),
                false, false, r, c);
        width = comp.getPreferredSize ().width;
        maxw = width > maxw ? width : maxw;
    }
    return maxw;
}
public static void main (String args []) {
    GJApp.launch (
        new Test ()."Setting Column Widths", 300, 300, 320, 140);
}

```

Swing 提示

使用 `JTable.getCellRenderer()` 和 `JTable.getCellEditor()`

使用 `JTable` 方法 `getCellRenderer (int row, int col)` 和 `getCellEditor (int row, int col)` 来获得特定表格单元的绘制器和编辑器。

虽然使用 `TableColumn` 方法 `getCellRenderer()` 和 `getCellEditor()` 能够从表格列中获得对表格绘制器和编辑器的引用，但是人们更喜欢使用 `JTable` 方法，因为如果没有明确设置一个列的绘制器和编辑器，那么 `JTable` 方法将返回缺省的绘制器和编辑器。

注意，`JTable` 中绘制器和编辑器访问方法的传送参数是行和列。缺省情况下，`JTable` 方法将返回相同的绘制器和编辑器给特定列所在的每一行，但它们可能由 `JTable` 子类重载以便返回一个绘制器给特定的表格单元。

类总结 19-3 总结了 `TableColumn` 类

类总结 19-3 `TableColumn`

扩展：`java.lang.Object`

实现：`java.io.Serializable`

1. 构造方法：

```

public TableColumn ()
public TableColumn (int modelIndex)
public TableColumn (int modelIndex, int width)
public TableColumn (int modelIndex, int width, TableCellRenderer, TableCellEditor)

```

若未指定，`modelIndex` 缺省值为 0，`width` 缺省值为 75，而 `renderer` 和 `editor` 为 `null`。`JTable` 给未明确设置的列提供了缺省的绘制器和编辑器。

2. 方法

(1) 单元编辑器

```

public TableCellEditor getCellEditor ()
public void setCellEditor (TableCellEditor)

```

当列被构造后，可在任一时间内设置表格列的单元编辑器。`cellEditor` 属性是一种简单属性，当设置单元编辑器时其属性变化事件并未激发。注意其属性与 `cellRenderer` 属性相反，`cell-`

Renderer 属性是一种关联属性^①。

(2) 单元绘制器

```
public TableCellRenderer getCellRenderer ()
public void setCellRenderer (TableCellRenderer)
```

单元绘制器也可在列构造后的任一时间内设置。cellRenderer 属性是一种关联属性，即当设置单元编辑器时将激发属性变化事件，它与 cellEditor 属性（非关联属性）相反。

(3) 头部绘制器

```
public TableCellRenderer getHeaderRenderer ()
public void setHeaderRenderer (Table cellRenderer)

protected TableCellRenderer createDefaultHeaderRenderer ()
```

与列单元不同，JTable 类未给列头部提供缺省的绘制器。因此，如果 TableColumn.setHeaderRenderer() 的参数是一个 null 绘制器，则这个方法会抛出一个异常，HeaderRenderer() 属性是一个关联属性。

createDefaultHeaderRenderer 方法创建了一个头部绘制器，缺省情况下表格列配备这个头部绘制器。这个绘制器扩展 DefaultTableCellRenderer（它扩展 JTable）并绘制从头部值的 toString 方法返回的值。

(4) 头部值

```
public Object getHeaderValue ()
public void setHeaderValue (Object)
```

表格列的头部值是列头部绘制器显示的一个对象。缺省情况下，列头部值利用表格列名字来进行初始化。HeaderValue 属性是一个关联属性。

(5) 属性变化监听器

```
public synchronized void addPropertyChangeListener (PropertyChangeListener)
public synchronized void removePropertyChangeListener (PropertyChangeListener)
```

当修改 TableColumn 的关联属性时将激发属性变化事件。以上列举的方法允许从表格列中添加和删除监听器。

(6) 首选宽度/最小宽度/最大宽度

```
public int getWidth ()
public int getMaxWidth ()
public int getMinWidth ()
public int getPreferredWidth ()
public void setMaxWidth (int)
public void setMinWidth (int)
public void setPreferredWidth (int)
public void setWidth (int)
```

以上方法是表格列的实际宽度、最小宽度、最大宽度和首选宽度的访问方法。

同组件一样，列也可以由另一对象（JTable）来调整大小。与组件不同，其最小、最大和首选取尺寸可被布局管理器忽略，当 JTable 类调整列宽度时，表格列的大小将不会比最小宽度更小，比最大宽度更大。

因为表格列由 JTable 类来调整大小，所以用 setWidth 方法来设置一个列的实际宽度通常是一种暂时的变化。

(7) 标识符/模型索引/重新调整大小

① CellEditor 属性在以后的 Swing 版本可能是关联的。

```

public Object getIdentifier ()
public int getModelIndex ()
public boolean getResizable ()

public void setIdentifier (Object)
public void setModelIndex (int)
public void setResizable (boolean)

```

以上列举的方法是表格列的标识符、模型索引和可调整性的访问方法。

注意 在 Swing 1.1 FCS 中，setResizable 方法不起作用，表格列总是可调整的。

(8) 实用方法

```

public void disableResizedPosting ()
public void enableResizedPosting ()

public void sizeWidthToFit ()

```

调用以上列举的前两种方法对表格列不起作用，这些方法仅在 Swing 中调用。

sizeWidthToFit 方法调整列的大小，以使列的宽度与列头部的首选宽度相匹配。

19.4 表格列模型

表格的列由一个实现 TableColumnModel 接口的对象来维护。表格列模型可以通过 JTable 类来访问并负责选取、添加、移动和删除表格列。

表格列模型还维护少量的属性，这些属性作为整体适用于表格列。例如，TableColumnModel.setColumnMargin() 设置各列之间的边距，在给定的表格中，所有列的边距都是相同的。

接口总结 19-2 总结了 TableColumnModel 接口

接口总结 19-2 TableColumnModel

1. 监听器登记

```

public abstract void addColumnModelListener (TableColumnModelListener)
public abstract void removeColumnModelListener (TableColumnModelListener)

```

表格列模型监听器接收列属性和选取变化的通知。以上列举的方法用于添加和删除监听器。

2. 添加/删除/移动列

```

public abstract void addColumn (TableColumn)
public abstract void removeColumn (TableColumn)

public abstract void moveColumn (int fromIndex, int toIndex)

```

可以从表格模型中添加和删除列。添加和删除列不会影响表格的数据，它仅影响那些显示在表格中的列。

以上列举的最后一个方法将列从一个索引移动到另一个索引。

3. 列选取

```

public abstract void setColumnSelectionAllowed (boolean)
public abstract boolean getColumnSelectionAllowed ()

public abstract int getSelectedColumnCount ()
public abstract int [] getSelectedColumns ()

public abstract ListSelectionModel getSelectionModel ()
public abstract void setSelectionModel (ListSelectionModel)

```

以上列举的前一个方法指定列选取操作是否能够进行。若方法 getColumnSelectionAllowed 返

回 true，则允许进行列选取操作，否则返回 false。

可以用方法 `getSelectedColumnCount` 和 `getSelectedColumns` 来获得当前选中的列数和它们的索引数组。

用以上列举的最后两个方法可以访问表格列模型所使用的列表选取模型。

4. 其他的列属性

```
public abstract TableColumn getColumn (int)
public abstract int getColumnCount ()
public abstract int getColumnIndex (Object)
public abstract int getColumnIndexAtX (int x)
public abstract int getColumnMargin ()
public abstract Enumeration getColumnns ()
public abstract int getTotalColumnsWidth ()
public abstract void setColumnMargin (int)
```

表格列模型提供了许多获取各种属性的方法，如给定索引的列、模型中的列数以及所有列占据的总宽度。

表格模型中仅 `columnMargin`，`selectionModel` 和 `columnSelectionAllowed` 属性是可以设置的。另外，这三个可设置的属性中，只有修改 `columnMargin` 属性才会激发事件^①。

19.4.1 DefaultTableColumnModel 类

Swing 只提供了 `TableColumnModel` 接口的一个实现：`DefaultTableColumnModel` 类。`DefaultTableColumnModel` 的类图见图 19-14。

`DefaultTableColumnModel` 扩展 `Object` 并实现 `TableColumnModel` 的接口。

`DefaultTableColumnModel` 将它的列保存在 `Vector` 中并包含有 `ListSelectionModel` 的一个实例。选取模型用于实现列选取操作，有关表和列选取的详细内容，请参见 19.5 节“表格选取”。

`DefaultTableColumnModel` 还实现 `PropertyChangeListener` 接口和 `ListSelectionListener` 接口，`PropertyChangeListener` 接口用于响应列中属性的变化，`ListSelectionListener` 接口用于检测列选取事件。列属性



图 19-14 DefaultTableColumnModel 的类图

① 所有的属性应该都是关联的，在 Swing 以后的版本中可能都是关联的。

变化事件和列选取事件由 TableColumnModelListener 进行监听。

DefaultTableColumnModel 用一个 boolean 变量指明是否允许列选取，并用一个整数值来表示列边距。

19.4.2 列边距

表格列的列边距表示一行中相邻两个单元之间的间隙。当修改列边距时，DefaultTableColumnModel 把一个 changeEvent 事件发送给已登记的 TableColumnModelListener 监听器。

图 19-15 所示的应用程序含有一个表格和一个控制列边距的滑杆。

图 19-15 上图显示应用程序的初始状态。缺省的列边距是 1 个像素。边距所占空间用于画网格的垂直线。当边距值是 0 个像素时，网格线不可见。

中图显示了 20 个像素的边距，因为只有单元背景是可以增亮的（边距不能），因此边距是明显可见的。

这个应用程序创建了一个具有简单表格模型的表格，该表格包裹在滚动窗格中，并作为中央组件添加到这个应用程序的内容窗格中。ControlPanel 的一个实例被创建并被作为上部组件添加到内容窗格中。

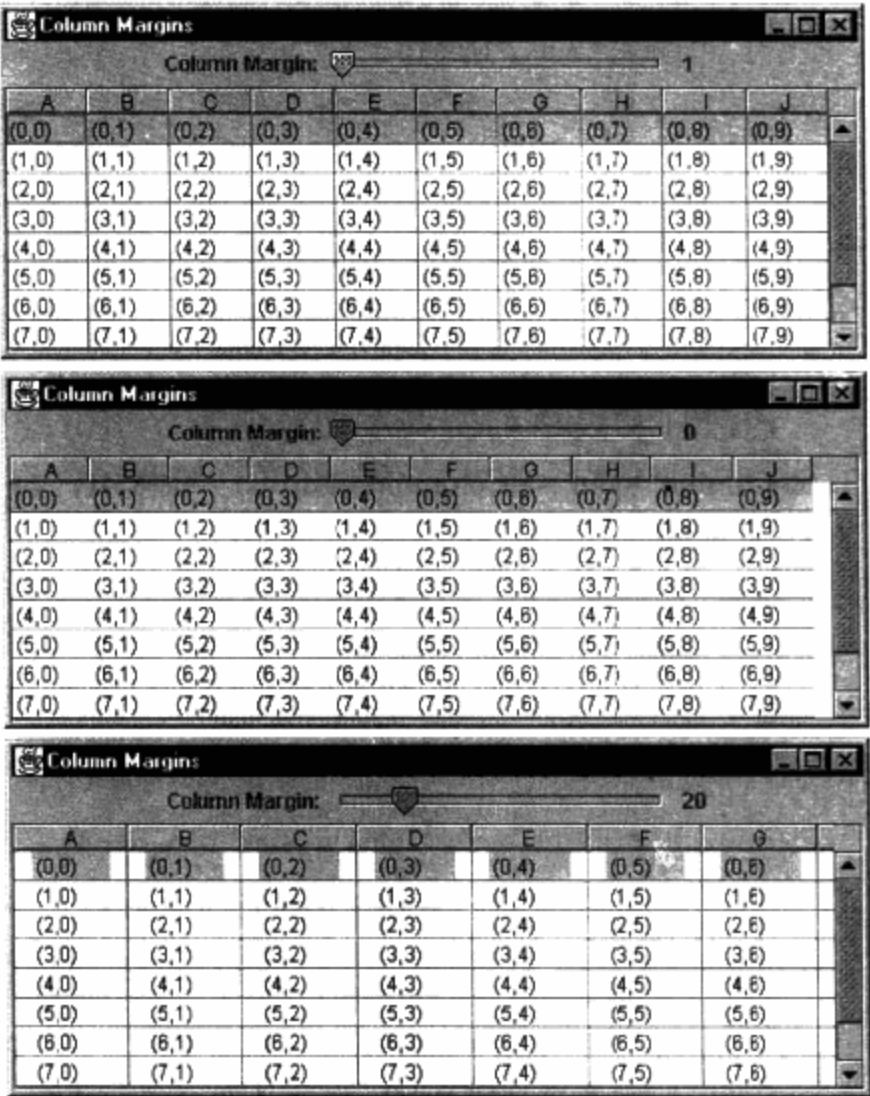


图 19-15 设置列边距

```
public class Test extends JFrame {
    JTable table = new JTable (
        new AbstractTableModel () {
            public int getRowCount () { return 10; }
            public int getColumnCount () { return 10; }

            public Object getValueAt (int row, int col) {
                return " (" + Integer.toString (row) + ", " +
                    Integer.toString (col) + ")";
            }
        }
    );

    public Test () {
        Container cp = getContentPane ();
        cp.add (new JScrollPane (table), BorderLayout.CENTER);
        cp.add (new ControlPanel (), BorderLayout.NORTH);
    }
}
```

这个控制面板包含有滑杆，它用缺省的列边距进行初始化。除滑杆外，这个控制面板还包含两个标签：一个标识这个滑杆，另一个显示这个滑杆的值。

```
...
class ControlPanel extends JPanel {
    ...
}
```

```

private JSlider slider = new JSlider (
    JSlider.HORIZONTAL, 0, 100,
    table.getColumnModel ().getColumnMargin ());

private JLabel label = new JLabel ();

public ControlPanel () {
    add (new JLabel ("Column Margin:"));
    add (slider);
    add (label);

    label.setText (
        Integer.toString (
            table.getColumnModel ().getColumnMargin ());
    ...

```

给滑杆和表格列模型添加监听器。当滑杆的值变化时，列边距被设置成滑杆的值。对 `TableColumnModel.setColumnMargin ()` 的调用，将导致表格列模型激发变化事件，并利用标签文本更新的方式进行处理。

```

...
slider.addChangeListener (new ChangeListener () {
    public void stateChanged (ChangeEvent e) {
        table.getColumnModel ().setColumnMargin (
            slider.getValue ());
    }
});

table.getColumnModel ().addColumnModelListener (
    new TableColumnModelListener () {
        public void columnMarginChanged (ChangeEvent e) {
            TableColumnModel m = table.getColumnModel ();
            label.setText (
                Integer.toString (m.getColumnMargin ());
        }

        // unfortunately, Swing does not have many
        // event adapter classes ...
        public void columnAdded (TableColumnModelEvent e) {
        }

        public void columnMoved (TableColumnModelEvent e) {
        }

        public void columnRemoved (
            TableColumnModelEvent e) {
        }

        public void columnSelectionChanged (
            ListSelectionEvent e) {
        }
    }
);

```

例 19-7 列出了图 19-15 所示应用程序的完整代码。

例 19-7 设置列边距

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.event. * ;

```

```

import javax.swing.table.*;
import java.util.*;

public class Test extends JFrame {
    JTable table = new JTable (
        new AbstractTableModel () {
            public int getRowCount () { return 10; }
            public int getColumnCount () { return 10; }

            public Object getValueAt (int row, int col) {
                return " (" + Integer.toString (row) + "," +
                    Integer.toString (col) + ")";
            }
        }
    );

    public Test () {
        Container cp = getContentPane ();
        cp.add (new JScrollPane (table), BorderLayout.CENTER);
        cp.add (new ControlPanel (), BorderLayout.NORTH);
    }

    class ControlPanel extends JPanel {
        private JSlider slider = new JSlider (
            JSlider.HORIZONTAL, 0, 100,
            table.getColumnModel () .getColumnMargin ());

        private JLabel label = new JLabel ();

        public ControlPanel () {
            add (new JLabel ("Column Margin:"));
            add (slider);
            add (label);

            label.setText (
                Integer.toString (
                    table.getColumnModel () .getColumnMargin ());

            slider.addChangeListener (new ChangeListener () {
                public void stateChanged (ChangeEvent e) {
                    table.getColumnModel () .setColumnMargin (
                        slider.getValue ());
                }
            });
        }

        table.getColumnModel () .addColumnModelListener (
            new TableColumnModelListener () {
                public void columnMarginChanged (ChangeEvent e) {
                    TableColumnModel m = table.getColumnModel ();
                    label.setText (
                        Integer.toString (m.getColumnMargin ());
                }
            }

            // unfortunately, Swing does not have many
            // event adapter classes ...
            public void columnAdded (TableColumnModelEvent e) {
            }

            public void columnMoved (TableColumnModelEvent e) {
            }

            public void columnRemoved (
                TableColumnModelEvent e) {
            }
        }
    }

```

```
        public void columnSelectionChanged (
                                ListSelectionEvent e) {
        }
    };
}

public static void main (String args []) {
    GJApp.launch (new Test (),
        "Column Margins", 150, 150, 500, 200);
}
}
```

19.4.3 隐藏列

表格列模型提供添加、移动和删除列的方法，图 19-16 所示的应用程序使用了所有的这些方法。该应用程序包含一个简单表格和一个复选框，这个复选框切换 First Name 列的可视性。

该应用程序创建一个 JTable 实例和一个控制面板，两者都被添加到这个应用程序的内容窗格中。

```
public class Test extends JFrame {
    JTable table = new JTable (
        new Object [] [] {
            {"Mouse", "Mighty", "M." },
            {"Mouse", "Polly", "A." },
            {"Doright", "Dudley", "L." }
        },
        new Object [] {
            "Last Name", "First Name", "Middle Initial"
        }
    );
    public Test () {
        Container cp = getContentPane ();
        cp.add (new JScrollPane (table), BorderLayout.CENTER);
        cp.add (new ControlPanel (), BorderLayout.NORTH);
    }
    ...
}
```

该控制面板包含这个复选框。把一个动作监听器加到这个复选框中，当取消选取这个复选框时，这个监听器删除列模型中的 FirstName 列。当选取这个复选框时，这个监听器把列添加到表格列模型中；而 TableColumnModel.addColumn 方法添加指定列作为最右列。因此，当把这个列添加到列模型中后，它从第三列（索引 2）移到了第二列（索引 1）。

```
...
class ControlPanel extends JPanel {
    private JCheckBox checkBox = new JCheckBox (
        "First Name Column Showing");
}
```

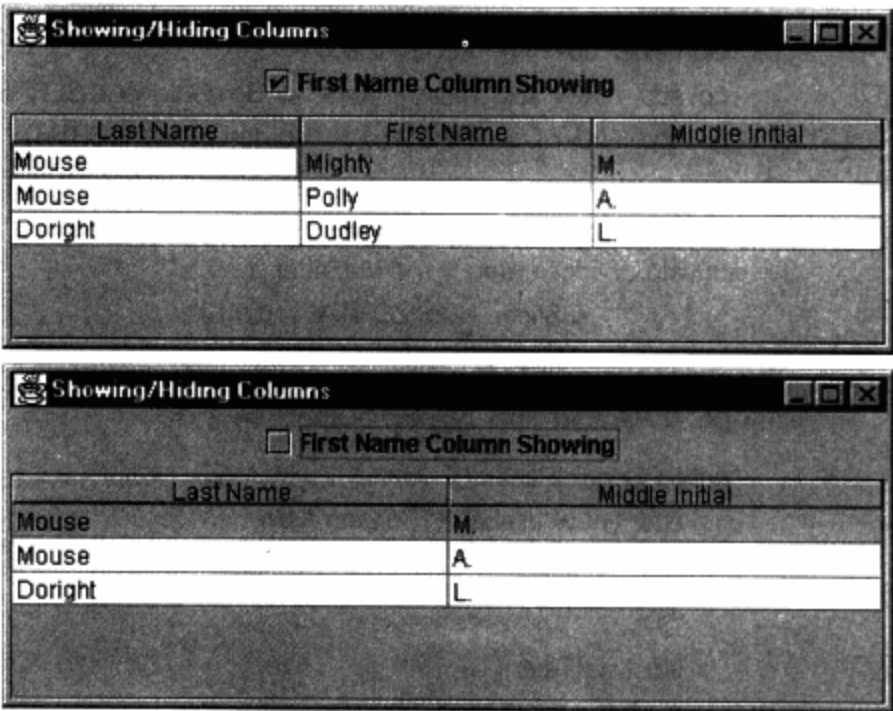


图 19-16 添加和删除列

```

public ControlPanel () {
    final TableColumnModel tcm = table.getColumnModel ();
    final TableColumn firstNameColumn =
        table.getColumnModel ("First Name");

    checkBox.setSelected (true);
    add (checkBox);

    checkBox.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent event) {
            if (checkBox.isSelected ()) {
                tcm.addColumn (firstNameColumn);
                tcm.moveColumn (2, 1);
            }
            else {
                tcm.removeColumn (firstNameColumn);
            }
            //Bug: the call to sizeColumnsToFit ()
            //should not be necessary
            table.sizeColumnsToFit (-1);
        }
    });
}
}

```

例图 19-8 列出了图 19-16 所示应用程序的完整代码。

例 19-8 添加和删除列

```

import javax.swing. * ;
import javax.swing.table. * ;
import java.awt. * ;
import java.awt.event. * ;
import java.util. * ;

public class Test extends JFrame {
    JTable table = new JTable (
        new Object [ ] [ ] {
            { "Mouse", "Mighty", "M." },
            { "Mouse", "Polly", "A." },
            { "Doright", "Dudley", "L." }
        },
        new Object [ ] {
            "Last Name", "First Name", "Middle Initial"
        }
    );

    public Test () {
        Container cp = getContentPane ();

        cp.add (new JScrollPane (table), BorderLayout.CENTER);
        cp.add (new ControlPanel (), BorderLayout.NORTH);
    }

    class ControlPanel extends JPanel {
        private JCheckBox checkBox = new JCheckBox (
            "First Name Column Showing");

        public ControlPanel () {
            final TableColumnModel tcm = table.getColumnModel ();
            final TableColumn firstNameColumn =

```

```
        table.getColumnModel ("First Name");

        checkBox.setSelected (true);
        add (checkBox);

        checkBox.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent event) {
                if (checkBox.isSelected ()) {
                    tcm.addColumn (firstNameColumn);
                    tcm.moveColumn (2, 1);
                }
                else {
                    tcm.removeColumn (firstNameColumn);
                }

                table.sizeColumnsToFit (-1);
            }
        });

        public static void main (String args []) {
            GJApp.launch (
                new Test (), "Showing/Hiding Columns", 300, 300, 450, 175);
        }
    }
```

19.4.4 锁定左边列

有时需要在水平方向上锁定表格最左边的列。例如，考虑图 19-17 所示的应用程序，它是一个电视节目单。其左边列可以被锁定，当节目单水平滚动时锁定的左边列仍然不变。

表格不支持锁定它们的列，所以要提供一个水平方向上锁定左边列的影像，第二个表格（此后作为头部表格）放入滚动窗格的行头部中，且把列头部放在了滚动窗格的左上角。限制头部表格的宽度与左边列相同，且不允许头部表格中的列进行排序。最后，从初始的表格列模型中删除左边列。

图 19-17 所示应用程序包括一个表格和一个复选框。选中复选框则锁定左边列，除去复选框的选取则不锁定左边列。图 19-17 中的上图显示左边列锁定前的情况，下图显示左边列锁定后的表格。两个表格都滚动到最右边。

这个应用程序创建了一个简单的表格模型，该表格模型由头部表格和主表格所共享。与主表格一样，头部表格有 10 列，但仅有第一列是可见的。

用主表格的列模型来获得对左边列的一个引用。

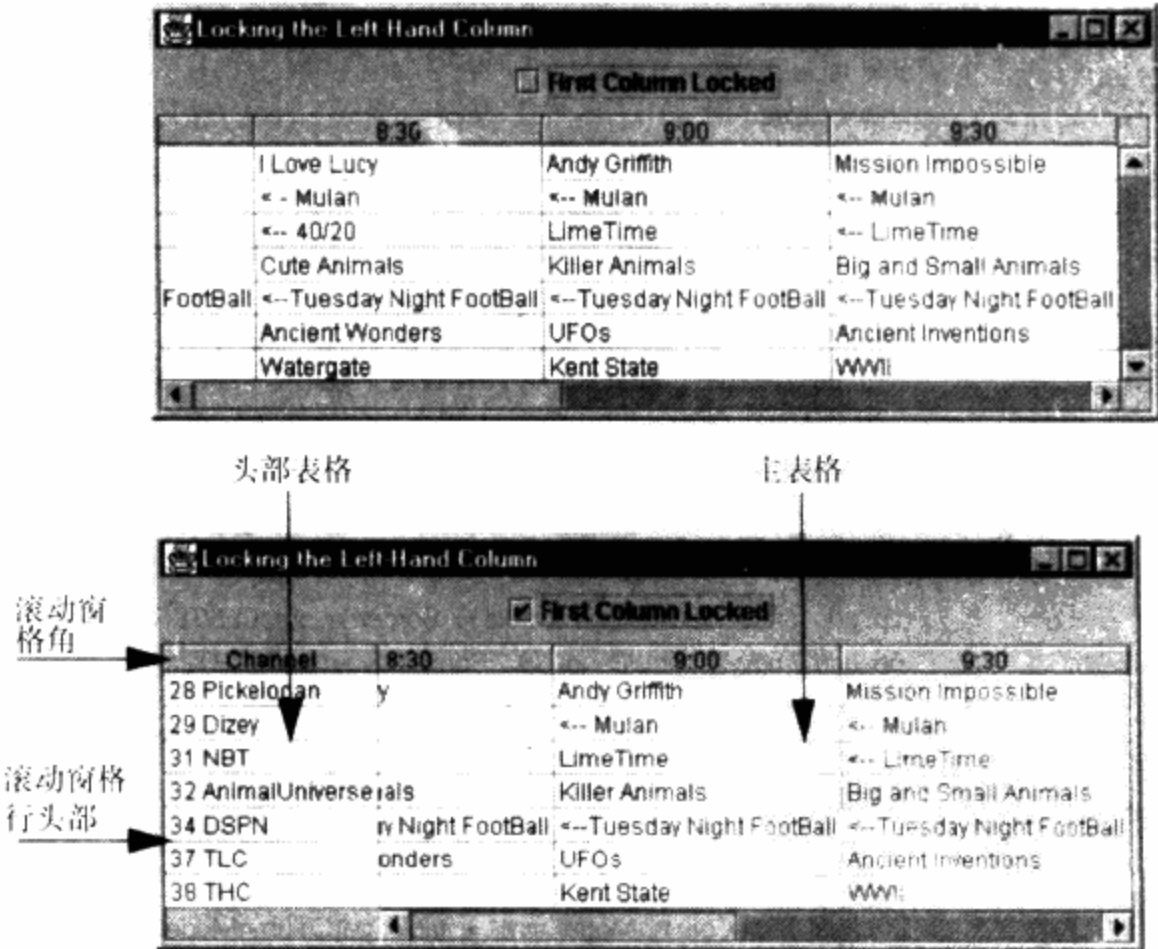


图 19-17 水平地锁定左边列

```

public class Test extends JFrame {
    class SharedModel extends AbstractTableModel {
        public int getRowCount () {return 10;}
        public int getColumnCount () {return 10;}
        public Object getValueAt (int row, int col) {
            return "(" + Integer.toString (row) + "," +
                Integer.toString (col) + ")";
        }
    }

    TableModel sharedModel = new SharedModel ();

    JTable table = new JTable (sharedModel),
        headerTable = new JTable (sharedModel);

    TableColumnModel tcm = table.getColumnModel ();
    TableColumn firstColumn = tcm.getColumn (0);
    ...
}

```

这两个表格的自动调整大小模式被设置为 `JTable.AUTO_RESIZE_OFF`，因此列尺寸根据它们的首选宽度来设置，当表格比它的窗格大时，主表格将设置一个滚动条。

如上所述，因为头部表格共享主表格模型，所以它也有 10 列。如果允许对头部表格重新排序，则被锁的列可拖动至右边而头部表格的第三列将成为第一列。为防止这种行为，头部表格不允许重新排序它的列。

头部表格的首选可滚动视口大小^①被设置为第一列的首选宽度加上其边距。如果边距空间不够，那么列右边的网格线将不可见，有关列边距的更多信息，请参见 19.4.2 节“列边距”。

```

...
public Test () {
    Container cp = getContentPane ();

    table.setAutoResizeMode (JTable.AUTO_RESIZE_OFF);
    headerTable.setAutoResizeMode (JTable.AUTO_RESIZE_OFF);

    headerTable.getTableHeader ().setReorderingAllowed (false);

    headerTable.setPreferredScrollableViewportSize (
        new Dimension (
            firstColumn.getPreferredWidth () +
            headerTable.getColumnModel ().getColumnMargin (),
            0));

    cp.add (new ControlPanel (), BorderLayout.NORTH);
    cp.add (new JScrollPane (table), BorderLayout.CENTER);
}
...

```

添加到这个应用程序复选框中的动作监听器利用 `SwingUtilities.getAncestorOfClass` 方法来获得对主表格滚动窗格的一个引用。有关 `SwingUtilities` 类的详细内容，请参见 6.3 节“Swing 实用工具”。

如果选中这个复选框，那么第一列将从主表格列模型中被删除而头部表格作为滚动窗格的行头部视图被安装。头部表格的表格头部被安装在滚动窗格的左上角。

如果这个复选框未选中，那么第一列被重新插入到主表格列模型中。因为列添加到了列模型所维护的列表中，所以第一列添加到表格的最右端。因此，该列从表的一端移到了另一端。

① 首选可滚动视口大小是表格的滚动窗格的首选大小。

最终滚动窗格的行头部视图将设置成 null 值。

```
...
class ControlPanel extends JPanel {
    JCheckBox checkBox = new JCheckBox ("First Column Locked");

    public ControlPanel () {
        add (checkBox);

        checkBox.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JScrollPane scrollPane = (JScrollPane)
                    SwingUtilities.getAncestorOfClass (
                        JScrollPane.class, table);

                if (checkBox.isSelected ()) {
                    tcm.removeColumn (firstColumn);

                    scrollPane.setRowHeaderView (headerTable);
                    scrollPane.setCorner (
                        JScrollPane.UPPER_LEFT_CORNER,
                        headerTable.getTableHeader ());
                }
                else {
                    int numCols = tcm.getColumnCount ();
                    tcm.addColumn (firstColumn);
                    tcm.moveColumn (numCols-1, 0);
                    scrollPane.setRowHeaderView (null);
                }
            }
        });
    }
}
```

例 19-9 列出了图 19-17 所示应用程序的完整代码。

例 19-9 锁定表格的左列

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.event. * ;
import javax.swing.table. * ;

public class Test extends JFrame {
    class SharedModel extends AbstractTableModel {
        public int getRowCount () {return 10;}
        public int getColumnCount () {return 10;}

        public Object getValueAt (int row, int col) {
            return " (" + Integer.toString (row) + "," +
                Integer.toString (col) + ")";
        }
    }

    TableModel sharedModel = new SharedModel ();
    JTable table = new JTable (sharedModel);
    JTable headerTable = new JTable (sharedModel);
    TableColumnModel tcm = table.getColumnModel ();
```

```

TableColumn firstColumn = tcm.getColumn (0);
public Test () {
    Container cp = getContentPane ();
    table.setAutoResizeMode (JTable.AUTO_RESIZE_OFF);
    headerTable.setAutoResizeMode (JTable.AUTO_RESIZE_OFF);
    headerTable.getTableHeader ().setReorderingAllowed (false);
    headerTable.setPreferredScrollableViewportSize (
        new Dimension (
            firstColumn.getPreferredWidth () +
            headerTable.getColumnModel ().getColumnMargin (),
            0));
    cp.add (new ControlPanel (), BorderLayout.NORTH);
    cp.add (new JScrollPane (table), BorderLayout.CENTER);
}
class ControlPanel extends JPanel {
    JCheckBox checkBox = new JCheckBox ("First Column Locked");
    public ControlPanel () {
        add (checkBox);
        checkBox.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                JScrollPane scrollPane = (JScrollPane)
                    SwingUtilities.getAncestorOfClass (
                        JScrollPane.class, table);
                if (checkBox.isSelected ()) {
                    tcm.removeColumn (firstColumn);
                    scrollPane.setRowHeaderView (headerTable);
                    scrollPane.setCorner (
                        JScrollPane.UPPER_LEFT_CORNER,
                        headerTable.getTableHeader ());
                }
                else {
                    tcm.addColumn (firstColumn);
                    int numCols = tcm.getColumnCount ();
                    tcm.moveColumn (numCols-1, 0);
                    scrollPane.setRowHeaderView (null);
                }
                table.revalidate ();
            }
        });
    }
}
public static void main (String args []) {
    GJApp.launch (
        new Test (), "Locking the Left-Hand Column",
        300, 300, 425, 210);
}

```

19.5 表格选取

可以用 `ListSelectionModel` 类支持的选取模式中的一种选取模式来选取表格行、列和单个单

元，这些选取模式如下所列：

- LstSelectionMode.SINGLE_SELECTION
- ListSelectionMode.SINGLE_INTERVAL_SELECTION
- ListSelectionMode.MULTIPLE_INTERVAL_SELECTION

JTable 和 TableColumnModel 分别有一个列表选取模型，可跟踪行、单元和列的选取状态。有关 ListSelectionMode 类的更多信息，请参见 19.2 节“列表模型”。

在表格中选取哪个对象（行、列或单元）是由下面的 JTable 方法控制的：

- JTable.setColumnSelectionAllowed (boolean)
- JTable.setRowSelectionAllowed (boolean)
- JTable.setCellSelectionEnabled (boolean)

图 19-18 所示的应用程序允许设置选取模式和表格中可选取的对象（行、列或单元）。图 19-18 中的所有表格都可以进行行、列选取，但单元除外。

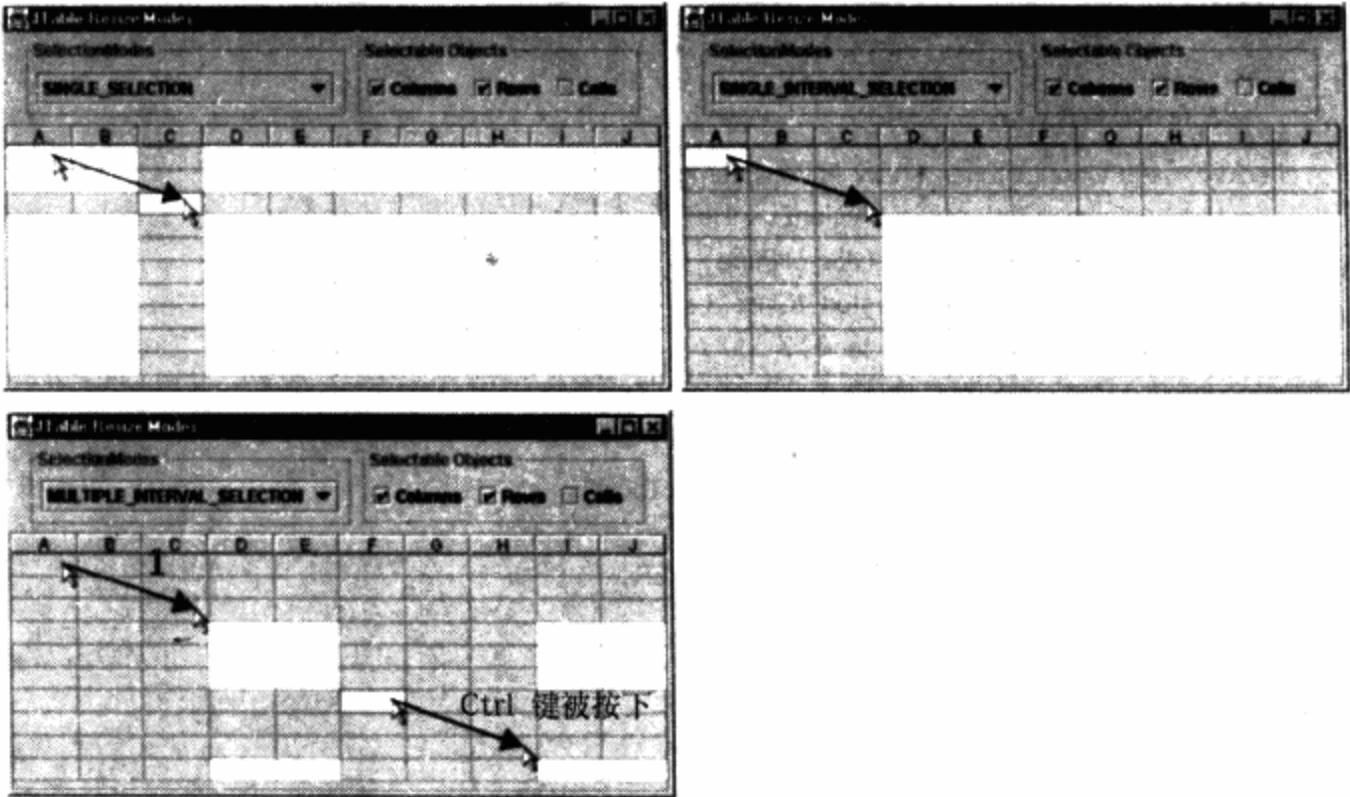


图 19-18 表格选取

图 19-18 的左上图显示了当把选取模式设置为单选取模式时按箭头方向拖动鼠标的结果。右上图显示了在单间隔选取模式中拖动鼠标时选取的行和列的情况。在下图显示了在多间隔选取模式下两次拖动鼠标的结果（第二次拖动鼠标是按住 Ctrl 键来执行的）。

这个应用程序创建了一个简单表格和一个 ControlPanel 实例——JPanel 的一个扩展列举如下。把表格和控制面板添加到内容窗格中：控制面板作为上部组件而表格作为中央组件。

```
public class Test extends JFrame {
    JTable table = new JTable (10, 10);

    public Test () {
        Container cp = getContentPane ();
        cp.add (new ControlPanel (), BorderLayout.NORTH);
        cp.add (new JScrollPane (table), BorderLayout.CENTER);
    }
    ...
}
```

ControlPanel 类包含两个面板，一个面板包含一个组合框，另一个面板包含三个复选框。两

个数组被实例化，一个数组含有字符串并显示在组合框中的，另一个数组代表 ListSelectionModel 类的相应选取常量。

```
class ControlPanel extends JPanel
    implements ActionListener, ItemListener {
    Object [] selectionModes = new Object [] {
        "SINGLE _ SELECTION",
        "SINGLE _ INTERVAL _ SELECTION",
        "MULTIPLE _ INTERVAL _ SELECTION",
    };
    int [] selectionConstants = {
        ListSelectionModel.SINGLE _ SELECTION,
        ListSelectionModel.SINGLE _ INTERVAL _ SELECTION,
        ListSelectionModel.MULTIPLE _ INTERVAL _ SELECTION,
    };

    JPanel selectionPanel = new JPanel (),
        selectablesPanel = new JPanel ()

    JCheckBox [] selectables = new JCheckBox [] {
        new JCheckBox ("Columns"),
        new JCheckBox ("Rows"),
        new JCheckBox ("Cells"),
    };
    JComboBox mode Combo = new JComboBox (selectionModes);
    ...
}
```

ControlPanel 构造方法设置了两个面板的边框并将组合框和复选框添加到各自的面板中。随后这两个面板添加到控制面板中并调用 private initializeControl 方法。

ControlPanel 类实现 ActionListener 和 ItemListener 接口，以便监听分别来自其复选框和组合框的选取。因此，这个控制面板添加到组合框中作为项监听器，并把这个控制面板添加到复选框中作为动作监听器。

```
...
public ControlPanel () {
    selectablesPanel.setBorder (
        BorderFactory.createTitledBorder (
            "SelectionModes"));
    selectionPanel.setBorder (
        BorderFactory.createTitledBorder (
            "Selectable Objects"));

    selectionPanel.add (mode Combo)

    for (int i=0; i < selectables.length; ++i) {
        selectablesPanel.add (selectables [i]);
        selectables [i].addItemListener (this);
    }

    initializeControls ();

    add (selectionPanel);
    add (selectablesPanel);

    modeCombo.addActionListener (this);
}
...
}
```

initializeControl 方法在表格当前的选取模式下选取相应的项来初始化组合框。复选框根据

允许的选取类型来进行初始化。

```
...
private void initializeControls () {
    int mode =
        table.getSelectionModel ().getSelectionMode ();
    if (mode == ListSelectionMode.SINGLE_SELECTION)
        modeCombo.setSelectedIndex (0);
    else if (mode ==
        ListSelectionMode.SINGLE_INTERVAL_SELECTION)
        modeCombo.setSelectedIndex (1);
    else if (mode ==
        ListSelectionMode.MULTIPLE_INTERVAL_SELECTION)
        modeCombo.setSelectedIndex (2);

    selectables [0] .setSelected (
        table.getColumnSelectionAllowed ());
    selectables [1] .setSelected (
        table.getRowSelectionAllowed ());
    selectables [2] .setSelected (
        table.getCellSelectionEnabled ());
}
...
```

当激发组合框中一个选项并被 `JTable.setSelectionModel()` 设置为表格选取模式时将调用方法 `ControlPanel.actionPerformed()`。

当选中复选框或取消选取时，方法 `ControlPanel.itemStateChanged()` 被调用。调用适当的 `JTable` 方法使复选框表示的选取类型有效。

```
...
public void actionPerformed (ActionEvent e) {
    int index = modeCombo.getSelectedIndex ();
    table.setSelectionMode (selectionConstants [index]);
}

public void itemStateChanged (ItemEvent e) {
    JCheckBox checkBox = (JCheckBox) e.getSource ();
    boolean b = checkBox.isSelected ();

    if (checkBox == selectables [0])
        table.setColumnSelectionAllowed (b);
    else if (checkBox == selectables [1])
        table.setRowSelectionAllowed (b);
    else if (checkBox == selectables [2])
        table.setCellSelectionEnabled (b);
}
}
```

例 19-10 列出了图 19-18 所示应用程序的完整代码。

例 19-10 表格选取

```
import javax.swing.*.*;
import javax.swing.event.*;
import javax.swing.table.*;
import java.awt.*;
```

```

import java.awt.event.*;
import java.util.*;

public class Test extends JFrame {
    JTable table = new JTable (10, 10);

    public Test () {
        Container cp = getContentPane ();

        cp.add (new ControlPanel (), BorderLayout.NORTH);
        cp.add (new JScrollPane (table), BorderLayout.CENTER);
    }
}

class ControlPanel extends JPanel
    implements ActionListener, ItemListener {
    Object [] selectionModes = new Object [] {
        "SINGLE _ SELECTION",
        "SINGLE _ INTERVAL _ SELECTION",
        "MULTIPLE _ INTERVAL _ SELECTION",
    };
    int [] selectionConstants = {
        ListSelectionModel.SINGLE _ SELECTION,
        ListSelectionModel.SINGLE _ INTERVAL _ SELECTION,
        ListSelectionModel.MULTIPLE _ INTERVAL _ SELECTION,
    };

    JPanel selectionPanel = new JPanel ();
    JPanel selectablesPanel = new JPanel ();

    JCheckBox [] selectables = new JCheckBox [] {
        new JCheckBox ("Columns"),
        new JCheckBox ("Rows"),
        new JCheckBox ("Cells"),
    };

    JComboBox modeCombo = new JComboBox (selectionModes);

    public ControlPanel () {
        selectionPanel.setBorder (
            BorderFactory.createTitledBorder (
                "SelectionModes"));
        selectablesPanel.setBorder (
            BorderFactory.createTitledBorder (
                "Selectable Objects"));

        selectionPanel.add (modeCombo);

        for (int i = 0; i < selectables.length; ++i) {
            selectablesPanel.add (selectables [i]);
            selectables [i].addItemListener (this);
        }

        initializeControls ();

        add (selectionPanel);
        add (selectablesPanel);

        modeCombo.addActionListener (this);
    }

    private void initializeControls () {
        int mode =
            table.getSelectionModel ().getSelectionMode ();
    }
}

```

```

        if (mode == ListSelectionModel.SINGLE_SELECTION)
            modeCombo.setSelectedIndex (0);
        else if (mode ==
            ListSelectionModel.SINGLE_INTERVAL_SELECTION)
            modeCombo.setSelectedIndex (1);
        else if (mode ==
            ListSelectionModel.MULTIPLE_INTERVAL_SELECTION)
            modeCombo.setSelectedIndex (2);

        selectables [0] .setSelected (
            table.getColumnSelectionAllowed ());
        selectables [1] .setSelected (
            table.getRowSelectionAllowed ());
        selectables [2] .setSelected (
            table.getCellSelectionEnabled ());
    }

    public void actionPerformed (ActionEvent e) {
        int index = modeCombo.getSelectedIndex ();
        table.setSelectionMode (selectionConstants [index]);
    }

    public void itemStateChanged (ItemEvent e) {
        JCheckBox checkBox = (JCheckBox) e.getSource ();
        boolean b = checkBox.isSelected ();

        if (checkBox == selectables [0])
            table.setColumnSelectionAllowed (b);
        else if (checkBox == selectables [1])
            table.setRowSelectionAllowed (b);
        else if (checkBox == selectables [2])
            table.setCellSelectionEnabled (b);
    }
}

public static void main (String args []) {
    GJApp.launch (
        new Test (), "JTable Resize Modes", 300, 300, 500, 290);
}
}

```

19.6 绘制和编辑

与 Swing 的其他有单元的组件相比，Swing 表格有单元值绘制器和编辑器（可选择）。

下面几节将列举一个应用程序来探讨表格单元的绘制和编辑。绘制和编辑几乎总是相关的且一起出现。在介绍应用程序使用的绘制器和编辑器前要先探讨一下这个应用程序的情况。图 19-19 所列的应用程序将涉及绘制器和编辑器，这些绘制器和编辑器将在后面介绍。

19.6.1 使用表格单元绘制器和编辑器

既使到 2000 年，大多数汽车音响商允许用一个特定的音响控制面板来预听音响，在这个音响控制面板上推动滑杆来选取音响。图 19-19 所示的表格模拟了这样一个音响控制面板。

通过在与音响相关联的 In Use 球状按钮上单击，可以从图 19-19 所示的表格中选取音响

(此后，这个表格称为音响面板表格)。这些球状按钮值是相互排斥的，若选取其中的一个球状按钮，则其他球状按钮就不被选取。同样，选取一行即选取了该行的 In Use 球状按钮，反之亦然。In Use 列有一个定制绘制器（扩展 DefaultTableCellRenderer）和一个实现 CellEditor 接口的编辑器。

Manufacturer 列和 Model 列是不可标记的（它们显示文本），且不可编辑，有缺省的绘制器。

只有商店职员才能改变 Price 列，所以该列是可编辑的，但该列的编辑有密码保护。当激活该列的编辑器时会显示一个密码对话框（参见图 19-27），如果输入密码正确，就允许以一种组合框的方式进行访问。

Dolby 列表示音响是否有杜比声音。因为音响特色不可改变，所以 Dolby 列是不可编辑的。另一方面 Bass 可以被打开和关闭。利用缺省的绘制器给 Dolby 和 Bass 列绘制 Boolean 值——一个复选框。Bass 列还配备一个缺省的编辑器。

在 Volume 列中的单元包含有一个滑杆和一个标签（跟踪滑杆的值）。标签和滑杆只有在它们所在的行被选中时才有效，双击标签可以重新设置音量。Volume 单元仅在它们的行被选中时才有效。

音响面板表格配备了 StereoDeckModel 的一个实例，StereoDeckModel 扩展 AbstractTableModel，例 19-11 中列出了 StereoDeckModel 类。

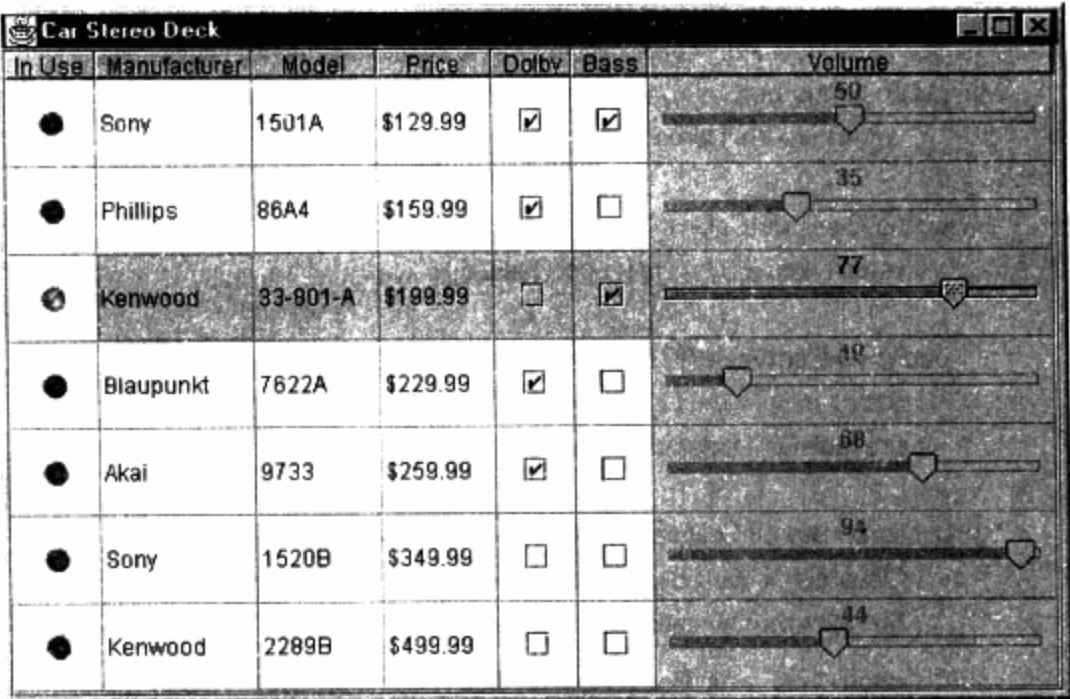


图 19-19 一个具有定制绘制器和编辑器的汽车音响面板

例 19-11 StereoDeckModel 类

```
class StereoDeckModel extends AbstractTableModel {
    String [] columnNames = {
        "In Use", "Manufacturer", "Model", "Price", "Dolby",
        "Bass", "Volume"
    };
    Object [] [] data = {
        { Boolean.FALSE, "Sony", "1501A",
          new Double (129.99), Boolean.TRUE,
          Boolean.TRUE, new Integer (50) },
        //rest of data omitted
    };
    public Object getValueAt (int row, int col) {
        return data [row] [col];
    }
    public int getRowCount () {
        return data.length;
    }
    public int getColumnCount () {
```

```

        return columnNames.length;
    }
    public String getColumnName (int col) {
        return columnNames [col];
    }
    public Class getColumnClass (int col) {
        return data [0] [col] .getClass ();
    }
    public void setValueAt (Object value, int row, int col) {
        data [row] [col] = value;
        fireTableCellUpdated (row, col);
    }
    public boolean isCellEditable (int row, int col) {
        Class cls = getColumnClass (col);
        String name = getColumnName (col);

        return (cls == Boolean.class && ! name.equals ("Dolby")) ||
            cls == Integer.class || cls == Double.class;
    }
    public void updateBulbs (int selectedRow) {
        for (int r=0; r < getRowCount (); ++r) {
            data [r] [0] = new Boolean (r == selectedRow);
        }
    }
}

```

该模型的数据由对象组所组成，大多数对象组已在例 19-11 清单中省略。这个模型把 In use, Price, Bass 和 Volume 列定义为可编辑单元。

注意，当调用重载 setValueAt 方法时激发一个表格单元更新事件。正如在本章 Swing 提示“AbstractTableModel 的扩展必须激发相应的事件”一节中所指出的那样，当修改数据时必须激发相应的事件。

模型还提供一个 updateBulbs 方法，该方法在第一列中实现相互排斥的选取（即选中一个球状按钮，则其他球状按钮就不能被选中），并给除选中行外其他的球状按钮赋值为 false。

图 19-19 所示的应用程序用 StereoDeckModel 的一个实例来创建一个表格。这个应用程序的构造方法初始化具有定制绘制器和编辑器的列。这个表格的列被调整了大小，列表的选取模式被设置成单选取模式。

表格选取模式的一个选取监听器更新左列中的球状按钮并在选取未调整之际重新绘制该表格。

```

public class Test extends JFrame {
    JTable table = new JTable (new StereoDeckModel ());
    public Test () {
        initializeInUseColumn ();
        initializePriceColumn ();
        initializeVolumeColumn ();
        sizeColumns ();
        table.setSelectionMode (
            ListSelectionModel.SINGLE_SELECTION);
        table.getSelectionModel ().addListSelectionListener (

```

```

        new ListSelectionListener () {
            public void valueChanged (ListSelectionEvent e) {
                StereoDeckModel model =
                    (StereoDeckModel) table.getModel ();

                if (! e.getValueIsAdjusting ()) {
                    model.updateBulbs (table.getSelectedRow ());
                    table.repaint ();
                }
            }
        };
        getContentPane ().add (new JScrollPane (table),
                                BorderLayout.CENTER);
    }
}

```

初始化具有定制绘制器和编辑器的列的方法列举如下。initializeInUseColumn 方法利用 bulb 绘制器和编辑器来设置 InUse 列。这个应用程序中所用到的绘制器和编辑器在随后的章节将讨论。

利用分别给 TableCellCurrencyRenderer 和 PriceEditor 实例设置列的绘制器和编辑器的方式，对 Price 列进行初始化。另外，用 Double 值对 PriceEditor 的组合框进行初始化和实例化。组合框的绘制器被配备给 ListCellCurrencyRenderer 的一个实例。

```

...
private void initializeInUseColumn () {
    TableColumn inUseColumn = table.getColumn ("In Use");
    inUseColumn.setCellRenderer (new BulbRenderer ());
    inUseColumn.setCellEditor (new BulbEditor ());
}

private void initializePriceColumn () {
    TableColumn priceColumn = table.getColumn ("Price");
    JComboBox combo = new JComboBox ();

    combo.addItem (new Double (159.99));
    combo.addItem (new Double (169.99));
    combo.addItem (new Double (229.99));
    combo.addItem (new Double (449.99));
    combo.addItem (new Double (699.99));

    combo.setRenderer (new ListCellCurrencyRenderer ());
    priceColumn.setCellRenderer (
        new TableCellCurrencyRenderer ());
    priceColumn.setCellEditor (new PriceEditor (combo));
}
...

```

Volume 列分别把它的绘制器和编辑器配备给 VolumeRender 和 VolumeEditor 的实例。表格的行高被配备给 Volume 列绘制器的高度。

```

...
private void initializeVolumeColumn () {
    TableColumn volumeColumn = table.getColumn ("Volume");
    TableCellRenderer renderer = new VolumeRenderer ();
    TableCellEditor editor = new VolumeEditor ();
    volumeColumn.setCellRenderer (renderer);
}

```

```

        volumeColumn.setCellEditor (editor);

        Dimension ps = ( (JPanel) renderer ).getPreferredSize ();
        table.setRowHeight (ps.height);
    }
    private void sizeColumns () {
        ...
    }
    ...
}

```

例 19-12 列出了图 19-19 所示应用程序的完整代码[⊖]。

例 19-12 使用绘制器和编辑器

```

import java.awt. * ;
import java.awt.event. * ;
import java.text. * ;
import java.util. * ;
import javax.swing. * ;
import javax.swing.event. * ;
import javax.swing.table. * ;

public class Test extends JFrame {
    JTable table = new JTable (new StereoDeckModel ());

    public Test () {
        initializeInUseColumn ();
        initializePriceColumn ();
        initializeVolumeColumn ();
        sizeColumns ();

        TableColumnModel tcm = table.getColumnModel ();
        TableCellEditor ce;

        table.setSelectionMode (
            ListSelectionModel.SINGLE_SELECTION);

        table.getSelectionModel ().addListSelectionListener (
            new ListSelectionListener () {
                public void valueChanged (ListSelectionEvent e) {
                    StereoDeckModel model =
                        (StereoDeckModel) table.getModel ();

                    if (! e.getValueIsAdjusting ()) {
                        model.updateBulbs (table.getSelectedRow ());
                    }
                }
            });

        getContentPane ().add (new JScrollPane (table),
            BorderLayout.CENTER);
    }

    private void initializeInUseColumn () {
        TableColumn inUseColumn = table.getColumnModel ().getColumn (0);
        inUseColumn.setCellRenderer (new BulbRenderer ());
        inUseColumn.setCellEditor (new BulbEditor ());
    }
}

```

⊖ 绘制器和编辑器类未在例 19-12 中列举。

```

|
private void initializePriceColumn () {
    TableColumn priceColumn = table.getColumnModel ("Price");
    JComboBox combo = new JComboBox ();

    combo.addItem (new Double (159.99));
    combo.addItem (new Double (169.99));
    combo.addItem (new Double (229.99));
    combo.addItem (new Double (449.99));
    combo.addItem (new Double (699.99));

    combo.setRenderer (new ListCellCurrencyRenderer ());
    priceColumn.setCellRenderer (
        new TableCellCurrencyRenderer ());
    priceColumn.setCellEditor (new PriceEditor (combo));
}

private void initializeVolumeColumn () {
    TableColumn volumeColumn = table.getColumnModel ("Volume");
    TableCellRenderer renderer = new VolumeRenderer ();
    TableCellEditor editor = new VolumeEditor ();

    volumeColumn.setCellRenderer (renderer);
    volumeColumn.setCellEditor (editor);

    Dimension ps = ( (JPanel) renderer ).getPreferredSize ();
    table.setRowHeight (ps.height);
}

private void sizeColumns () {
    //listing omitted for brevity's sake.
}

public static void main (String args []) {
    GJApp. launch (
        new Test (), "Car Stereo Deck", 300, 300, 559, 368);
}

|
class ListCellCurrencyRenderer extends DefaultListCellRenderer {
    public Component getListCellRendererComponent (
        JList list,
        Object value,
        int index,
        boolean isSelected,
        boolean hasFocus) {

        JLabel c = (JLabel)
            super.getListCellRendererComponent (
                list, value, index,
                isSelected, hasFocus);

        Format format = NumberFormat.getCurrencyInstance ();
        c.setText (value == null ? "" : format.format (value));
        return c;
    }
}

|
class TableCellCurrencyRenderer extends DefaultTableCellRenderer
|
    public void setValue (Object value) {
        Format format = NumberFormat.getCurrencyInstance ();

```

```
setText (value == null ? "" : format.format (value));
```

19.6.2 表格单元绘制器

与其他的 Swing 绘制器一样，表格单元绘制器被定义成一个接口，该接口只定义返回组件的方法。接口总结 19-3 总结了 TableCellRenderer 接口。

接口总结 19-3 TableCellRenderer

TableCellRenderer 接口只定义了一个方法，这个方法返回一个组件：

```
public abstract Component getTableCellRendererComponent (JTable table, Object value,
    boolean isSelected, boolean cellHasFocus, int row, int col)
```

像橡皮图章那样，由 getTableCellRendererComponent 返回的组件用于绘制到表格单元中。这个方法以表格、要绘制的值、值所在的行与列和这个单元是否被选取或是否有焦点为参数。

Swing.table 包以 DefaultTableCellRenderer 类的形式提供了一个缺省的绘制器，DefaultTableCellRenderer 扩展 JLabel 并实现 TableCellRenderer 接口。

如果一个标签正好与一个绘制器的组件相匹配，那么它通常扩展 DefaultTableCellRenderer 而不是直接实现 TableCellRenderer 接口，如表 19-6 所指出的那样。

表 19-6 表格单元绘制器

绘制器	类/接口	功能	当 ... 使用/实现/或扩展
TableCellRenderer	接口	提供对绘制器组件的访问	绘制器组件不是一个标签
DefaultTableCellRenderer	类	提供对绘制器组件、前景背景颜色和设置值的访问	绘制器组件是一个标签

图 19-19 所示应用程序的音量绘制器如图 19-20 所示。VolumeRenderer 类扩展 JPanel 并包含代表音量大小的标签和滑杆。

因为 VolumeRenderer 类是一个面板而不是一个标签，所以 TableCellRenderer 接口被直接实现而不是利用 DefaultTableCellRenderer 的扩展来实现。例 19-13 列出了 VolumeRenderer 类。

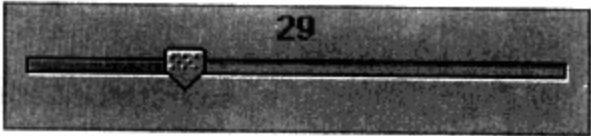


图 19-20 音量绘制器

VolumeRenderer 类初始化滑杆的首选大小，并指定 JSlider.isFilled 客户属性（以金属视感效果来填充滑杆左边的滑动槽）。有关 JSlider.isFilled 客户属性的详细内容，请参见 11.2.1 节“填充滑杆”。

因为标签的水平排列和水平文本位置都设置为 JLabel.CENTER，所以这个标签的文本放在中间。把这个标签添加到这个面板中作为这个面板的上部组件，而指定滑杆为这个面板的中央组件。

最后，把一个变化监听器添加到滑杆中，以便保持标签文本与滑杆的同步。

例 19-13 VolumeRenderer

```

import java.awt. * ;
import javax.swing. * ;
import javax.swing.event. * ;
import javax.swing.table. * ;

class VolumeRenderer extends JPanel
    implements TableCellRenderer {
    private JSlider slider = new JSlider ();
    private JLabel label = new JLabel ("value");

    public VolumeRenderer () {
        slider.setOrientation (SwingConstants.HORIZONTAL);
        slider.setPreferredSize (new Dimension (200, 30));
        slider.putClientProperty ("JSlider.isFilled", Boolean.TRUE);

        label.setHorizontalAlignment (JLabel.CENTER);
        label.setHorizontalTextPosition (JLabel.CENTER);

        setLayout (new BorderLayout ());
        add (label, BorderLayout.NORTH);
        add (slider, BorderLayout.CENTER);

        slider.addChangeListener (new ChangeListener () {
            public void stateChanged (ChangeEvent e) {
                label.setText (
                    Integer.toString (slider.getValue ());
                );
            }
        });
    }
    ...
}

```

从方法 `getVolumeRendererComponent` 返回的组件是这个绘制器本身（是一个面板）。这个滑杆和标签值都被设置，并根据是否选取单元来设置它们的允许状态。如果未选中音量单元，那么这个标签和滑杆是无效的，参见图 19-19。

`VolumeRenderer` 类提供标签和滑杆的访问方法。该访问方法由音量编辑器所使用，参见 19.6.5 节。

```

...
public Component getTableCellRendererComponent (
    JTable table, Object value,
    boolean isSelected,
    boolean hasFocus,
    int row, int col) {
    Integer v = (Integer) value;
    slider.setValue (v.intValue ());
    label.setText (v.toString ());

    slider.setEnabled (isSelected);
    label.setEnabled (isSelected);

    return this;
}

public JSlider getSlider () {

```



```
return slider;
}

public JLabel getLabel () {
    return label;
}
```

19.6.3 DefaultTableCellRenderer 类

图 19-21 示出了 DefaultTableCellRenderer 类的类图，DefaultTableCellRenderer 类是 Swing 提供的唯一的表格单元绘制器。

DefaultTableCellRenderer 扩展 JLabel 并维护对一个边框和未选取的前景色和背景色的引用。这些颜色用于重新设置选取后绘制器的前景色和背景色。

图 19-22 显示了图 19-19 表格中的两行。左列中的球状按钮由 BulbRenderer 的一个实例来绘制，例 19-14 列出了该实例。如果一行被选中，那么球状按钮的 boolean 值被设置成 true 且以增亮图标来绘制这个球状按

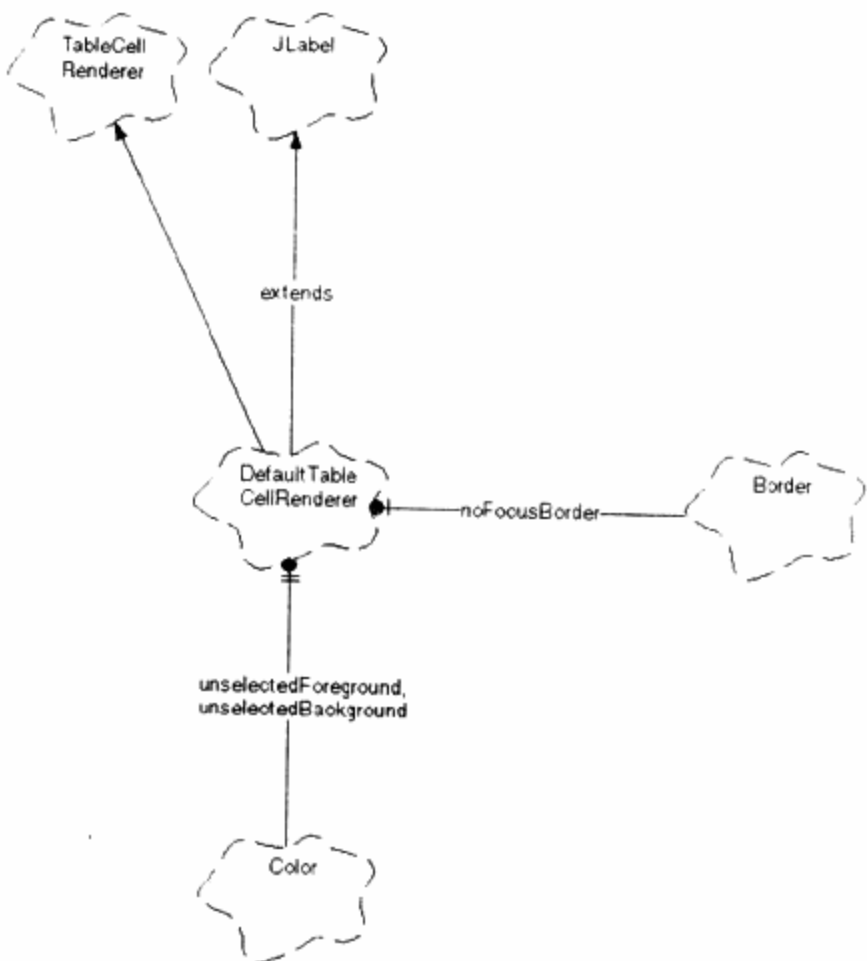


图 19-21 DefaultTableCellRenderer 类的类图

	Kenwood	33-801-A	\$199.99	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
	Blaupunkt	7622A	\$229.99	<input checked="" type="checkbox"/>	<input type="checkbox"/>	

图 19-22 球状按钮绘制器

钮。相反，如果行未被选中，那么其值设置成 false，这个球状按钮用较暗的图标来绘制。

BulbRenderer 类的实现比 VolumeRenderer 类更简单（参见例 19-13），主要因为 BulbRenderer 扩展 DefaultTableCellRenderer 类。因为 DefaultTableCellRenderer 实现 JTable 类，所以 BulbRenderer 的实例是标签。BulbRenderer 构造方法设置水平排列，以便这个标签的内容（即球状按钮）被居中放置。

getTableRendererComponent 方法根据单元值是 true 或 fales 来设置这个标签的图标。对音量绘制器（和在这种情况下的大多数其他的绘制器）而言，getTableRendererComponent 返回绘制器本身。

例 19-14 BulbRenderer

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

class BulbRenderer extends DefaultTableCellRenderer {
    private ImageIcon darkBulb = new ImageIcon ("button.jpg"),
        brightBulb = new ImageIcon ("button_lit.jpg");

    public BulbRenderer () {
```

```

        setHorizontalAlignment (JLabel.CENTER);
    }
    public Component getTableCellRendererComponent (
        JTable table, Object value,
        boolean isSelected,
        boolean hasFocus,
        int row, int col) {
        Boolean b = (Boolean) value;
        setIcon (b.booleanValue () ? brightBulb : darkBulb);
        return this;
    }
}

```

19.6.4 表格格式化绘制器

DefaultTableCellRenderer 类实现 public setValue 方法，这个方法设置绘制器的值。

根据单元是否被选中或是有焦点来设置绘制器的边框、颜色和字体，然后从绘制器的 getTableCellRendererComponent 方法中调用 DefaultTableCellRenderer.setValue ()。

DefaultTableCellRenderer.setValue () 提供了一个方便钩，用这个钩来格式化在表格单元中显示的数据。通过扩展 DefaultTableCellRenderer 并重载 setValue ()，可以用许多办法来修改绘制器的值。例如，图 19-19 所示表格的 Price 列配备有一个绘制器，它重载 setValue () 以便将一个 Double 值格式化成当前值，这个绘制器如下所列：

```

class TableCellCurrencyRenderer
    extends DefaultTableCellRenderer {
    public void setValue (Object value) {
        Format format = NumberFormat.getCurrencyInstance ();
        super.setValue (value == null ? "" : format.format (value));
    }
}

```

TableCellCurrencyRenderer 用 java.text.NumberFormat 的一个实例来格式化其值，然后将格式化的值传送给 DefaultTableCellRenderer.setValue ()。

Swing 提示

格式化绘制器可以重载 DefaultTableCellRenderer.setValue ()

DefaultTableCellRenderer 扩展 JLabel，因此从 getTableCellRendererComponent () 返回对自身的一个引用。在返回这个引用之前，先根据是否选取表格单元或是有焦点来设置这个绘制器的边框、颜色和字体，然后再调用 DefaultTableCellRenderer.setValue ()。

DefaultTableCellRenderer.setValue 方法将这个绘制器的文本设置成字符串，该字符串由单元值的 toString 方法返回，如下所示：

```

//from DefaultTableCellRenderer.java
protected void setValue (Object value) {
    setText ((value == null) ? "" : value.toString ());
}

```

setValue 方法提供一个方便钩，它允许 DefaultTableCellRenderer 扩展重载值的显示方式。例如，如同“表格的格式化绘制器”讨论的那样，不管绘制器的边框、颜色或字体怎么样，格式

化绘制器都可以重载 `setValue()` 来格式化值。

类总结 19-4 总结了 `DefaultTableCellRenderer` 类

类总结 19-4 `DefaultTableCellRenderer`

扩展: `JLabel`

实现: `TableCellRenderer`

1. 构造方法:

```
public DefaultTableCellRenderer ()
```

由 `DefaultTableCellRenderer` 类提供的唯一的构造方法创建了一个内衬为 (1, 2, 1, 2) 的空边框。把该边框安装作为这个绘制器的边框, 并调用方法 `setOpaque(true)` 将绘制器设置成不透明状态。

2. 方法

```
Public Component getTableCellRendererComponent (JTable,
                                                Object value,
                                                boolean isSelected,
                                                boolean hasFocus,
                                                int row, int column)
```

```
protected void setValue (Object)
public void setBackground (Color)
public void setForeground (Color)
public void updateUI ()
```

根据是否选中绘制器的组件或有焦点, `getTableCellRendererComponent` 设置绘制器的颜色, 字体和边框, 然后调用 `setVaule()`。

`DefaultTableCellRendererComponent` 简单地将 `toString` 方法返回的字符串设置成绘制器显示的文本, 但是 `DefaultTableCellRendererComponent` 的扩展可能希望重载 `setValue()` 方法以执行像格式化那样的操作。

从 `JLabel` 类中重载 `setBackground` 和 `setForeground` 方法来存储颜色, 因此在绘制器颜色设置成选取色后还可以恢复其原来的颜色。从 `JLabel` 类中重载 `UpdateUI` 方法可以设置前景色和背景色。

19.6.5 单元编辑器

单元编辑器具有下面的功能:

- 提供用于编辑单元值的一个组件。
- 当编辑取消或停止时通知 `CellEditorListener`。
- 确定单元是否可编辑和 (或) 可选取。
- 返回一个编辑值

与绘制器一样, 编辑器需要有一个组件。与绘制器组件不同, 编辑器的组件作为一种虚拟的橡皮图章来绘制单元, 它实际上安装在一个单元中并用来编辑一个值。

表格和树的编辑器都实现 `CellEditor` 接口, 接口总结 19-4 列出了 `CellEditor` 接口定义的方法。

接口总结 19-4 `CellEditor`

1. 单元编辑器的监听器

```
public abstract void addCellEditorListener (CellEditorListener)
public abstract void removeCellEditorListener (CellEditorListener)
```

当取消编辑或停止时，将通知 `CellEditorListener`。上面所列的这些方法允许向单元编辑器登记监听器。

2. 编辑状态

```
public abstract void cancelCellEditing ()
public abstract boolean stopCellEditing ()
```

`CancelCellEditing` 方法可以无条件地停止编辑和恢复编辑值。

`StopCellEditing` 方法请求编辑器停止编辑，该请求仅当 `stopCellEditing ()` 返回值为 `true` 时才有效。

3. 单元的可编辑性/可选取性

```
public abstract boolean isCellEditable (EventObject)
public abstract boolean shouldSelectCell (EventObject)
```

`JTable` 类使用 `isCellEditable` 方法和 `TableModel.isCellEditable ()` 方法共同决定单元是否可编辑。如果模型和单元编辑器的方法 `isCellEditable ()` 都返回 `true` 值，那么认为单元是可编辑的。

方法 `shouldSelectCell` 决定事件是否应该选取与编辑器相关联的单元。遗憾的是，这个功能（使用编辑器决定哪个单元是可选取的）使单元在没有编辑器的情况下不可选取是不可能的。

4. 单元值

```
public abstract Object getCellEditorValue ()
```

方法 `getCellEditorValue` 返回编辑器的值。通常，`getCellEditorValue` 方法不是由开发人员直接调用而是在适当时间由方法 `JTable.editingStopped ()` 来调用。

19.6.6 表格单元编辑器

注意，除提供一个组件之外，`CellEditor` 还定义了与单元编辑器功能（列举在“单元编辑器”中）相匹配的方法。这些功能留给 `CellEditor` 接口的扩展来完成，这些功能就是 `TableCellEditor` 和 `TreeCellEditor`。有关树编辑和 `TreeCellEditor` 类，请参见 20.7 节“树单元编辑”。

接口总结 19-5 总结了 `TableCellEditor` 接口。

接口总结 19-5 `TableCellEditor`

```
Public abstract Component getTableCellEditorComponent (JTable table, Object value,
boolean isSelected, int row, int col)
```

表格单元编辑器除实现由 `CellEditor` 接口定义的方法外，还必须提供一个组件供编辑时使用。对绘制器而言，由 `TableCellEditor.getTableCellEditorComponent ()` 返回的组件是编辑器本身。

1. 安装编辑器组件

为了有效地实现表格编辑器，有必要了解安装表格编辑器和修改编辑器值时事件发生的顺序。前者的说明见图 19-23。

理论上，任何类型的事件都可以初始化编辑，但缺省时，当 `JTable.editCellAt ()` 检测到鼠标按下事件时，`BasicTableUI` 启动滚动的球。

如果表格中的一个单元正在被编辑，那么利用 `CellEditor.stopCellEditing` 方法可以请求编辑器停止编辑。编辑器的 `stopCellEditing` 方法返回 `false` 值，表明编辑器未停止编辑，这个表格从 `edit-`

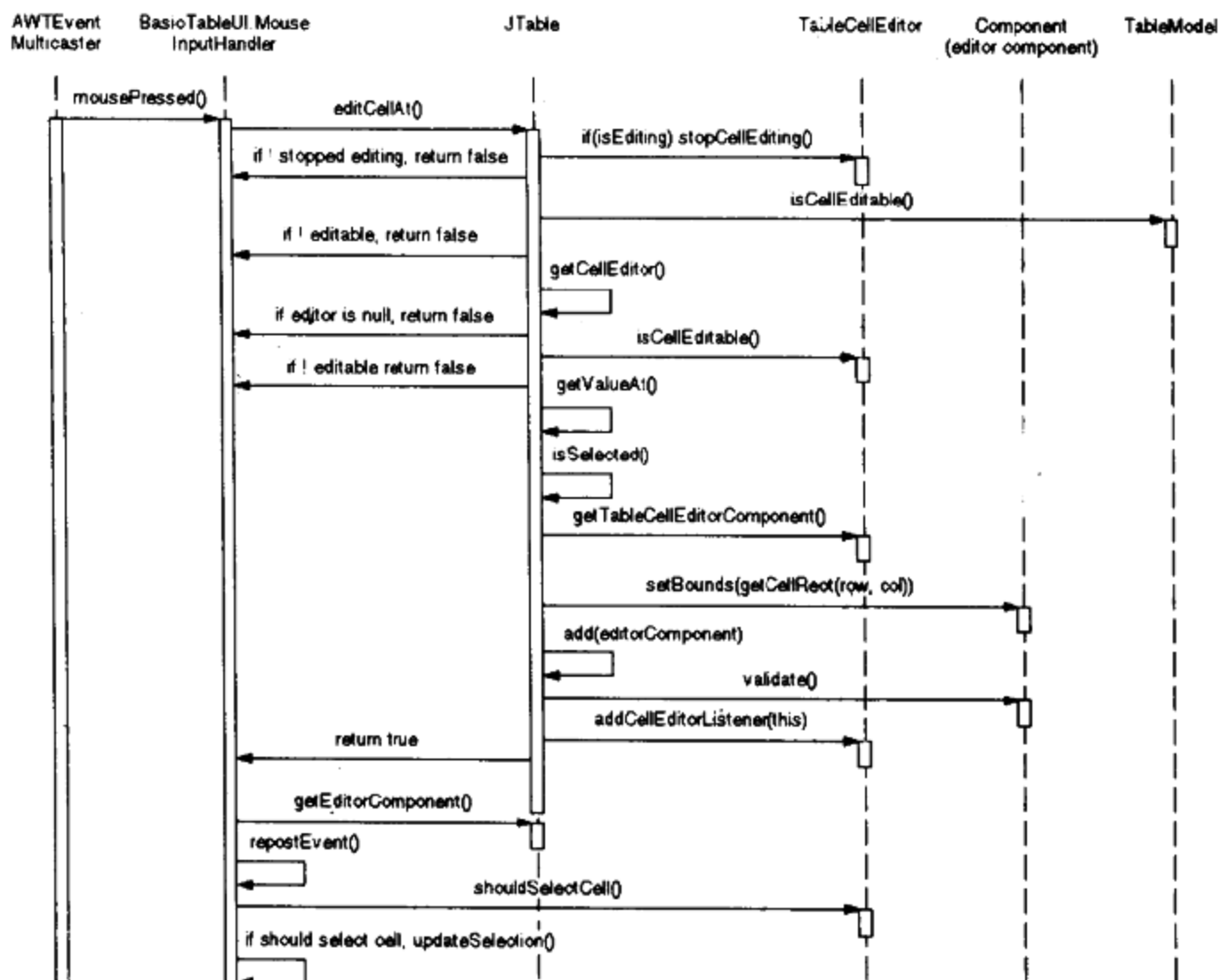


图 19-23 安装编辑器组件

CellAt 方法中返回。否则，表格询问它的模型和编辑器，询问其中的单元值是否是可编辑的。

如果表格模型和编辑器同意这个单元是可编辑的，那么表格就获得这个单元值并确定是否选取单元中，且两者被传送给编辑器的 getTableCellEditorComponent 方法。

表格一旦具有编辑器的组件，它就可以将组件的边框设置成由单元占据的矩形，并将组件添加到表格中。如果该组件有效，可以调整其大小并填充到单元中。最后，表格作为单元编辑器的监听器向编辑器登记并从 editCellAt 方法返回 true 值。

表格将编辑器的组件安装到相应的单元后，表格的 UI 代表将鼠标按下事件传送给编辑器的组件。即引起一个与编辑器安装相同的鼠标按下事件传送给这个编辑器的组件。

当把鼠标事件传送给新安装的编辑器后，表格的 UI 代表调用编辑器的 shouldSelectCell 方法以找出是否应该选取单元所在的行。如果 shouldSelectCell 返回 true，那么用 UI 代表的方法来选取这个单元所在的行。

2. 设置编辑值

图 19-23 显示了当初始化编辑时，方法调用的先后顺序。图

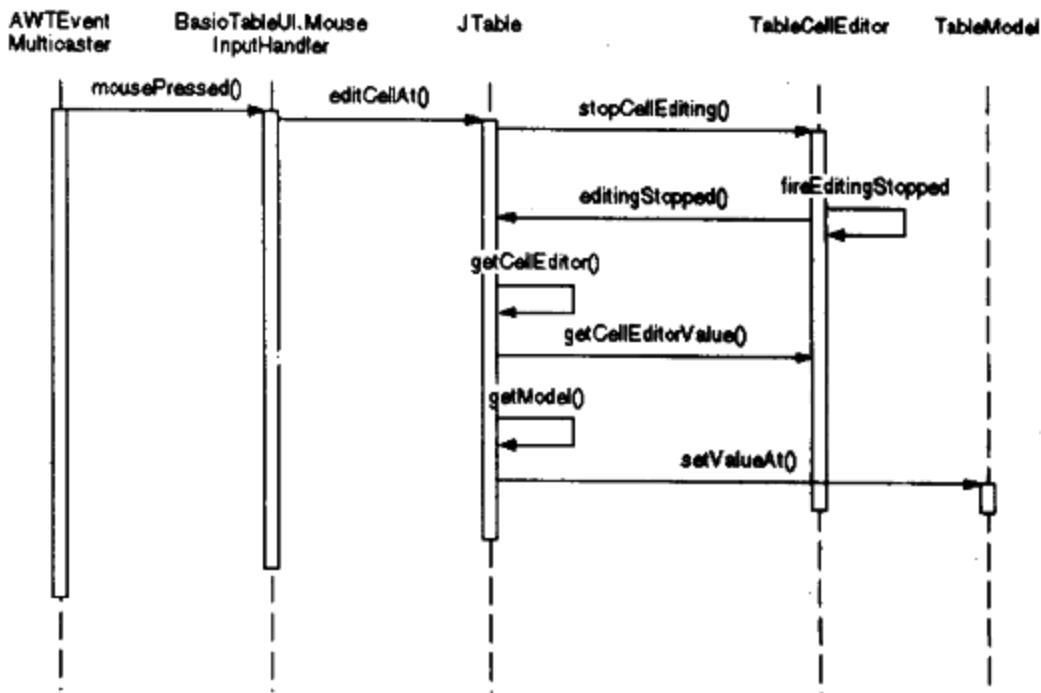


图 19-24 设置编辑值

19-24 显示了停止编辑时，方法调用的顺序。

如“安装编辑器组件”所讨论的那样，一个鼠标按下事件导致调用 `JTable.editCellAt()` 方法，请求当前活动编辑器停止编辑。如果编辑器停止，那么就把一个变化事件发送给所有向编辑器登记的 `CellEditorListener`。

回想图 19-23，当把一个编辑器安装在表格中时，这个表格作为 `CellEditorListener` 向这个编辑器进行登记。因此，当编辑器激发停止编辑的变化事件时，表格通过获取这个编辑器的值来响应，这个编辑器的值被传送给这个表格模型的 `setValueAt` 方法。

19.6.7 实现 TableCellEditor 接口

因为 Swing 不提供抽象编辑器类，该抽象编辑器类封装有 `TableCellEditor` 要求的一般内部处理的事物，所以实现 `TableCellEditor` 接口比实际要困难得多。[⊖] 例如，用于登记单元监听器和激发事件的方法（这些方法在 `CellEditor` 中定义）本质上是样板代码，必须为直接实现 `TableCellEditor` 接口的表格单元编辑器实现这个样板代码。

`DefaultCellEditor` 类实现这个功能，但 `DefaultCellEditor` 被限制直至其有效地扩展。

有关 `DefaultCellEditor` 类的更多信息，请参见“`DefaultCellEditor`”。

因为 Swing 不提供抽象单元编辑器类，所以下面将提供 `AbstractCellEditor` 的一个实现。

1. AbstractCellEditor

`AbstractCellEditor` 类实现 `TableCellEditor` 接口和 `TreeCellEditor` 接口。例 19-15 列出了 `AbstractCellEditor` 类。

例 19-15 AbstractCellEditor

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import javax.swing.tree.*;
import java.awt.event.MouseEvent;
import java.util.EventObject;

abstract public class AbstractCellEditor
    implements TableCellEditor {
    protected EventListenerList listenerList =
        new EventListenerList ();

    protected Object value;
    protected ChangeEvent changeEvent = null;
    protected int clickCountToStart = 1;
    ...
}
```

`AbstractCellEditor` 使用一个 `EventListenerList` 来维护 `CellEditorListener` 的一个列表，并提供两个 `protected` 方法把停止编辑和取消编辑事件发送给监听器。

```
...
public void addCellEditorListener (CellEditorListener l) {
    listenerList.add (CellEditorListener.class, l);
}
```

⊖ 接口--> 抽象类---> 缺省类术语在 Swing 中广泛使用，但本例中未有。

```

    |
    |
    | public void removeCellEditorListener (CellEditorListener l) {
    |     listenerList.remove (CellEditorListener.class, l);
    | }
    |
    | protected void fireEditingStopped () {
    |     Object [] listeners = listenerList.getListenerList ();
    |     for (int i = listeners.length-2; i >= 0; i -= 2) {
    |         if (listeners [i] == CellEditorListener.class) {
    |             if (changeEvent == null)
    |                 changeEvent = new ChangeEvent (this);
    |             ((CellEditorListener)
    |                 listeners [i+1]).editingStopped (changeEvent);
    |         }
    |     }
    | }
    |
    | protected void fireEditingCanceled () {
    |     Object [] listeners = listenerList.getListenerList ();
    |     for (int i = listeners.length-2; i >= 0; i -= 2) {
    |         if (listeners [i] == CellEditorListener.class) {
    |             if (changeEvent == null)
    |                 changeEvent = new ChangeEvent (this);
    |             ((CellEditorListener)
    |                 listeners [i+1]).editingCanceled (changeEvent);
    |         }
    |     }
    | }
    |
    | ...

```

AbstractCellEditor 维护一个代表编辑器值的 object 引用。为了方便，AbstractCellEditor 除实现 TableCellEditor.getCellEditorValue 方法外还提供了一个值的设置方法。

```

...
public Object getCellEditorValue () {
    return value;
}
public void setCellEditorValue (Object value) {
    this.value = value;
}
...

```

与 DefaultCellEditor 一样，AbstractCellEditor 提供一个 clickCountToStart 属性，它指出开始编辑所需的鼠标点击数。

```

...
public void setClickCountToStart (int count) {
    clickCountToStart = count;
}
public int getClickCountToStart () {
    return clickCountToStart;
}
...

```

如果传送给 isCellEditable () 的是一个鼠标事件并且点击数等于或大于 ClickCountToStart，AbstractCellEditor，那么 AbstractCellEditor 定义一个单元是可编辑的且所有的单元是可选取的。

```

...
public boolean isCellEditable (EventObject anEvent) {

```



```
if (anEvent instanceof MouseEvent) {
    if ( ((MouseEvent) anEvent).getClickCount() <
        clickCountToStart)
        return false;
    return true;
}

public boolean shouldselectCell (EventObject anEvent) {
    return true;
}

...
}
```

StopCellEditing 和 cancelCellEditing 方法激发相应的事件，当 StopCellEditing () 返回 true 时，表示编辑被停止。

```
...
public boolean stopCellEditing () {
    fireEditingStopped ();
    return true;
}

public void cancelCellEditing () {
    fireEditingCanceled ();
}

...
}
```

AbstractCellEditor 类未实现 TableCellEditor 定义的 getTableCellEditoComponent 方法。Abstract-CellEditor 的扩展必须实现 getTableCellEditorComponent ()。

2. 扩展 AbstractCellEditor

图 19-25 所示的 AbstractCellEditor 类可以用于简化定制单元编辑器的实现。例如，图 19-19 的 volume 列中的选定行的编辑器扩展 AbstractCellEditor。^①

图 19-25 显示了单个音量编辑器。上图在激活编辑器时立即显示编辑器组件，中图显示了把滑杆的控制块拖动到新位置时编辑器的情况。下图描述了在编辑器标签中双击时的情况，此时取消编辑并重新设置这个编辑器的值。

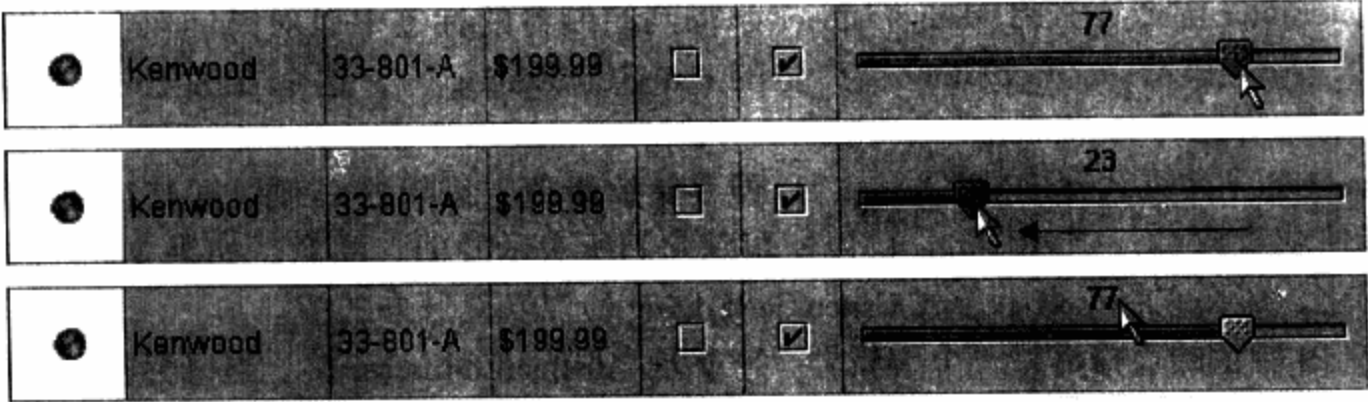


图 19-25 在编辑器的标签中双击将取消这次编辑

例 19-16 列出了 VolumeEditor 类。

例 19-16 VolumeEditor

```
import java.awt.*;
```

① AbstractCellEditor 不是 Swing 的一个组件，参见“AbstractCellEditor”。

```

import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;

class VolumeEditor extends AbstractCellEditor {
    VolumeRenderer renderer = new VolumeRenderer ();

    public VolumeEditor () {
        renderer.getLabel ().addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                if (e.getClickCount () == 2)
                    cancelCellEditing ();
            }
        });
    }

    public Component getTableCellEditorComponent (
        JTable table, Object value,
        boolean isSelected,
        int row, int column) {
        JSlider slider = renderer.getSlider ();
        slider.setValue ( ((Integer) value).intValue ());
        return renderer;
    }

    public boolean stopCellEditing () {
        JSlider slider = renderer.getSlider ();

        setCellEditorValue (new Integer (slider.getValue ()));

        return super.stopCellEditing ();
    }
}

```

单元编辑器组件通常就是一个单元绘制器组件的副本。[⊖] 例如，Volume 编辑器显示了一个滑杆和一个标签，它们也是音量绘制器的组件。但编辑器很少利用绘制器进行编辑，VolumeEditor 类可以利用方法 getTableCellEditorComponent 返回 VolumeRenderer 的一个实例。

VolumeEditor 构造方法将 MouseListener 添加到绘制器的标签中以对鼠标双击做出反应，即取消编辑。调用由 AbstractCellEditor 类产生的 cancelCellEditing ()，这个方法把一个相应的事件发送给所有已登记的 CellEditorListener。

VolumeEditor 类重载 stopCellEditing () 以便存储编辑值。编辑器必须根据其组件的操作不断更新编辑值或重载 CancelCellEditing () 以便跟踪它们所表示的值。注意，如果 VolumeEditor 类响应滑杆变化，那么它将不得不重载 CancelCellEditing () 以恢复原来的值。

图 19-19 所示的表格有另一个编辑器，它扩展 AbstractCellEditor 类。当鼠标在一个单元按下时，左列的球状按钮由灰暗变成闪亮。这个转换过程由 BulbEditor 的实例来完成，如下所示：

```

class BulbEditor extends AbstractCellEditor {
    BulbRenderer renderer = new BulbRenderer ();

    public BulbEditor () {
        renderer.addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {

```

⊖ 并不总是这样，具有组合框编辑器的单元，经常不是由组合框绘制。

遗憾的是, 由于一个表格错误, 在编辑器被安装后立即停止编辑会导致弹出错误信息, 这个错误信息由表格的 UI 代表来消除。如果把对 `fireEditingStopped()` 的调用放在 `Runnable` 中进行且传送给 `SwingUtilities.invokeLater()`, 这个错误信息就可以避免, 因为代码是在下一次事件周期中运行的。有关 `SwingUtilities.invokeLater` 方法的更多信息, 请参见 2.4 节“Swing 和线程”。

3. DefaultCellEditor 类

Swing 提供了 `TableCellEditor` 接口的一个实现, 即 `DefaultCellEditor` 类, 这个类提供了一个构造编辑器的便利途径, 这个编辑器有一个文本域、一个复选框或一个组合框, 它们被作为这个编辑器的组件。`DefaultCellEditor` 类提供三个构造方法, 其中每个构造方法都有一个文本域, 一个复选框或一个组合框参数。

`DefaultCellEditor` 类实现一个名为 `EditorDelegate` 的 `protected` 嵌套类, 并且每个 `DefaultCellEditor` 类维护一个对 `EditorDelegate` 的引用。`DefaultCellEditor` 类的实例代表对一个 `EditorDelegate` 实例的响应。例如, 如下面这样实现 `DefaultCellEditor.getCellEditorValue()` 方法:

```
//From DefaultCellEditor.java...
public Object getCellEditorValue () {
    return delegate.getCellEditorValue ();
}
```

当请求一个 `DefaultCellEditor` 类的实例值时, 它返回由它的代表提供的值。其他 `DefaultCellEditor` 方法也按这种方式来进行。

每个 `DefaultCellEditor` 构造方法启动 `EditorDelegate` 的一个内部类扩展, 这个扩展被设计成组件类型并被传送给这个构造方法。例如, `DefaultCellEditor(JTextField)` 创建一个重载 `EditorDelegate.getCellEditorValue()` 的代表, 以便返回包含在文本域内的文本。因此, 对于用一个文本域构造的缺省单元编辑器, `DefaultCellEditor.getCellEditorValue()` 返回文本域的文本。同样, `DefaultCellEditor(JCheckBox)` 返回一个 `boolean` 值, 表示是否选取了复选框。对于利用一个复选框构造的缺省单元编辑器来说, 编辑器将从方法 `DefaultCellEditor.getCellEditorValue()` 返回一个 `boolean` 值。

图 19-26 显示的是 `DefaultCellEditor` 类的类图。

因为 `DefaultCellEditor` 扩展 `Object` 并实现 `TableCellEditor` 和 `TreeCellEditor` 接口, 所以缺省单元

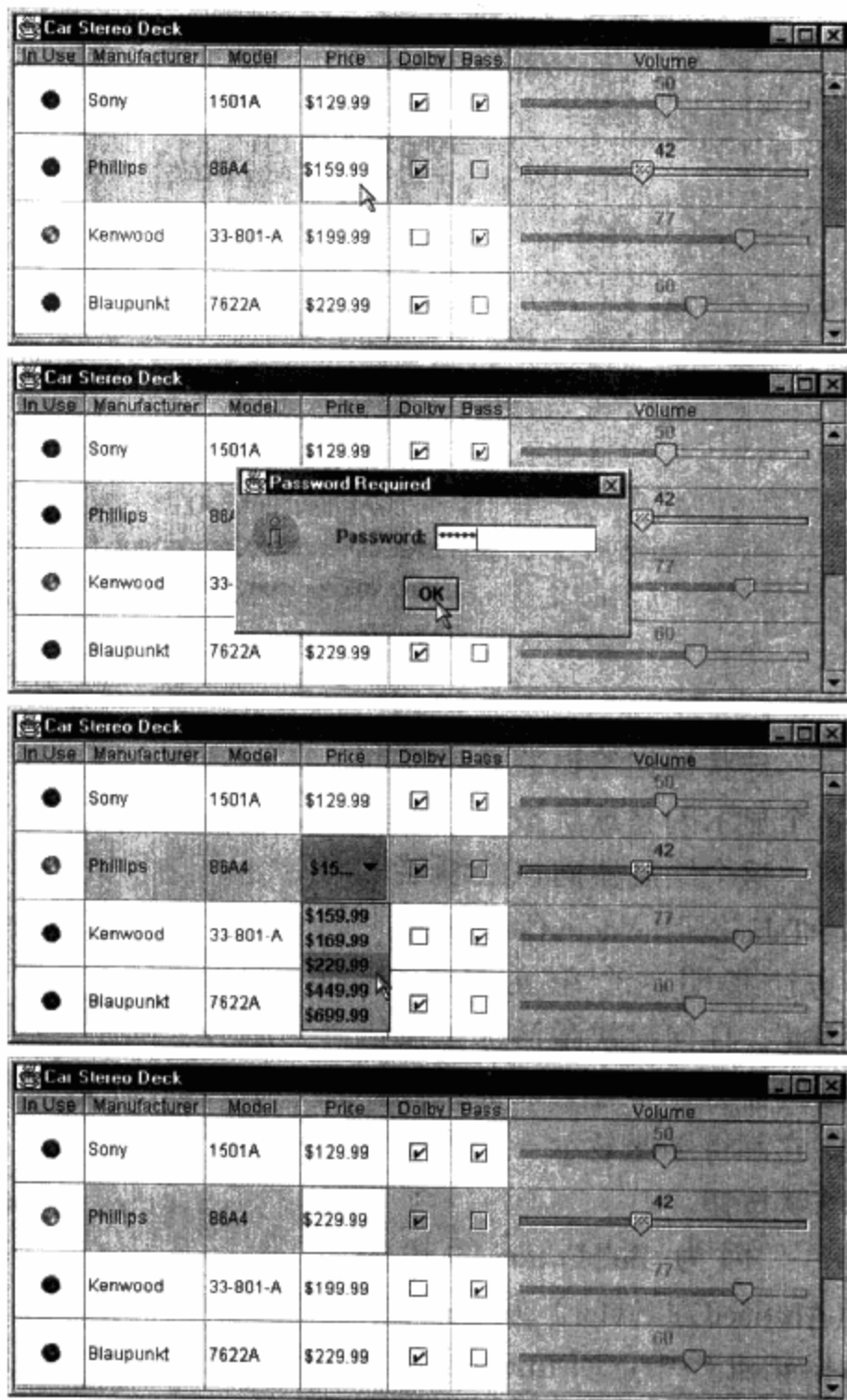


图 19-27 一个密码保护的可编辑单元

编辑器能够用于表格和树。

DefaultCellEditor 类维护对组件和编辑器代表的引用。DefaultCellEditor 也维护一个 integer 值，这个值代表开始编辑所需的鼠标点击数。当编辑已停止或取消时，EventListenerList 通知单元编辑器的监听器。

图 19-19 所示表格中的 Price 列配备了一个 PriceEditor 实例和一个 DefaultCellEditor 扩展。PriceEditor 构造方法以一个组合框为参数，把这个组合框传送给 DefaultCellEditor 的构造方法。

price 列的单元是否可编辑依赖于用户是否能够提供一个密码。从上到下，图 19-27 所示的是 Price 列中的单元正在编辑的情况。当鼠标在单元中点击时，就会出现一个密码对话框。如果输入的密码正确，就出现一个编辑的组合框；如果密码不正确，就会结束编辑会话。

注意 一个表格错误将造成密码框被连续显示两次。

例 19-17 列出了 PriceEditor 类

例 19-17 列出了 PriceEditor 类

```
import javax.swing.*;
import java.util.EventObject;

class PriceEditor extends DefaultCellEditor {
    public PriceEditor (JComboBox combo) {
        super (combo);
    }

    public boolean isCellEditable (EventObject e) {
        JPanel messagePanel = new JPanel ();
        JPasswordField pwf = new JPasswordField (10);

        messagePanel.add (new JLabel ("Password:"));
        messagePanel.add (pwf);

        JOptionPane.showMessageDialog (null,
            messagePanel, "Password Required",
            JOptionPane.INFORMATION_MESSAGE);

        if (pwf.getText ().equals ("dolby")) {
            return true;
        }
        else {
            JOptionPane.showMessageDialog (null,
                "Wrong Password!", "Access Failed",
                JOptionPane.INFORMATION_MESSAGE);
            return false;
        }
    }
}
```

PriceEditor 类重载 isCellEditable () 以便显示一个含有密码域的消息对话框。如果密码正确，那么这个方法返回 true 值且将编辑单元。如果密码不正确，则 isCellEditable () 将返回 false 值。

类总结 19-5 总结了 DefaultCellEditor 类

类总结 19-5 DefaultCellEditor

扩展：Object

实现：TableCellEditor

1. 构造方法

```
public DefaultCellEditor (JCheckBox)
public DefaultCellEditor (JComboBox)
public DefaultCellEditor (JTextField)
```

DefaultCellEditor 提供三个构造方法，每个构造方法都以一个代表组件为参数，参见“DefaultCellEditor 类”。

如果 DefaultCellEditor 类提供无参数的构造参数，它就更加可以重复使用，因为它可以利用 DefaultCellEditor.EditorDelegate 进行扩展并用于任何组件。事实上，一个 DefaultCellEditor 的简单无参数构造方法大大消除了对 AbstractCellEditor 类的需要，并给扩展类生成了基本的功能。

2. 方法

(1) CellEditor 方法

```
public void addCellEditorListener (CellEditorListener)
public void removeCellEditorListener (CellEditorListener)

public boolean isCellEditable (EventObject)
public boolean shouldSelectCell (EventObject)

public boolean cancelCellEditing ()
public boolean stopCellEditing ()

public Object getCellEditorValue ()
```

除以上列举的前两种方法外，所有由 CellEditor 接口定义的方法由 DefaultCellEditor 用编辑器的代表来实现。

如果鼠标点击数等于 ClickCountToStart 属性值且编辑器的代表认为单元是可编辑的，那么 DefaultCellEditor 也认为单元是可编辑的。

如果鼠标点击够数，那么 DefaultCellEditor 由安装在编辑器中的鼠标按下事件来选取。另外，DefaultCellEditor.shouldSelectCell() 调用这个代表的 startCellEditing 方法。

stopCellEditing() 和 cancelCellEditing() 激发变化事件发送给已登记的 CellEditorListener。stopCellEditing 方法调用这个代表的 stopCellEditing 方法。

(2) 树/表格编辑器组件

```
Public Component getTableCellEditorComponent (JTable, Object, boolean, int, int)
Public Component getTreeCellEditorComponent (JTree, Object, boolean, boolean, boolean, int)
```

由 TableCellEditor 和 TreeCellEditor 接口定义的方法返回传送给 DefaultCellEditor 构造方法的组件。由上面所列的两个方法配置的组件分别用于表格和树。

(3) 点击数

```
public int getClickCountToStart ()
public void setClickCountToStart (int)
```

ClickCountToStart 属性决定安装一个编辑器所需的鼠标连续点击次数。

(4) 组件

```
public Component getComponent ()
```

当安装一个界面样式时，JTable.updateUI() 调用 DefaultCellEditor.getComponent() 方法以便调用组件的 updateUI() 方法。

getComponent 方法不在 CellEditor 接口中定义，因而，JTable.updateUI() 查看这个编辑器是否是 DefaultCellEditor 的一个实例。如果是，就调用 DefaultCellEditor.getComponent()。所有这些将是无关的，除非扩展 DefaultCellEditor 比以前要困难得多，这种扩展适用于全部界面样式。

(5) 激发事件

```
protected void fireEditingCanceled ()
protected void fireEditingStopped ()
```

当编辑停止或取消时，DefaultCellEditor 提供两个 protected 方法把事件发送给已登记的 CellEditorListener。

19.7 表格行

与表格列不同，表格行的地位较低，Swing 不提供表格行的类。因此，JTable 类和表格模型给出大量操作行的方法。

19.7.1 行高

表格的行高由 JTable.setRowHeight 方法设置。经常要求将表格的行高设置成最高单元的行高，见图 19-28 所示的表格。

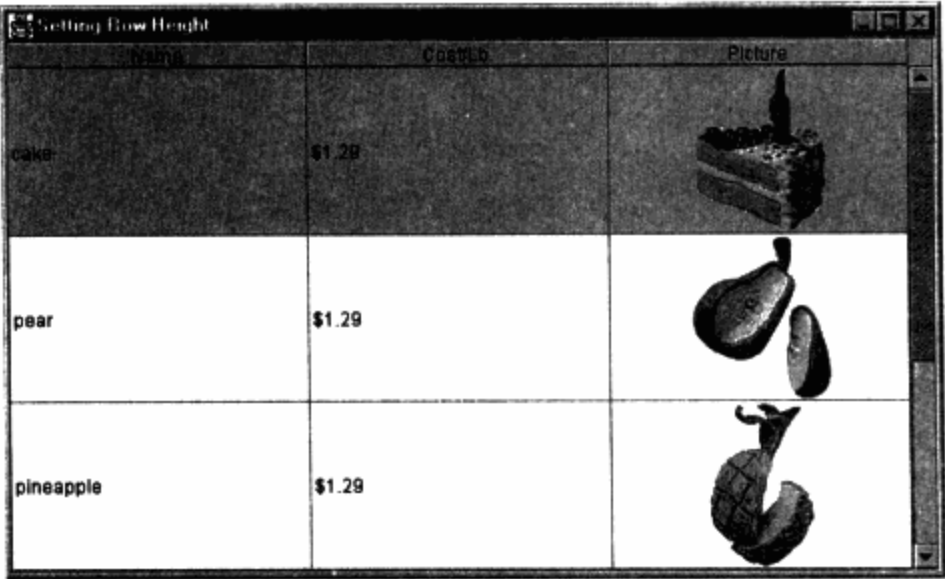


图 19-28 设置行高

图 19-28 所示的应用程序创建一个表格并用字符串和图标进行填充。这个应用程序还实现了返回该表格中单元最大行高的 getMaxRowHeight 方法。

对于表格中的每一行，getMaxRowHeight 方法调用 getMaxRowHeightForColumn ()，以便返回单个列中单元的最大行高。

```
public class Test extends JFrame {
    ...
    public Test () {
        table.setRowHeight (getMaxRowHeight ());
        ...
    }
    ...
    public int getMaxRowHeight () {
        int columnCount = table.getColumnCount (), h = 0, maxh = 0;
        for (int i = 0; i < columnCount; ++i) {
            TableColumn column =
                table.getColumnModel ().getColumn (i);

            h = getMaxRowHeightForColumn (column);
            maxh = Math.min (h, maxh);
        }
        return maxh;
    }
    ...
}
```

getMaxRowHeightForColumn 方法获得对列绘制器的一个引用，用这个引用来获得对这个绘制器组件的一个引用。对于列中的每个单元，这个绘制器组件的最大高度被计算，且该方法返回最高组件的高度。

```
...
public int getMaxRowHeightForColumn (TableColumn column) {
```



```

int height = 0, maxh = 0, c = column.getModelIndex ();
for (int r = 0; r < table.getRowCount (); ++r) {
    TableCellRenderer renderer =
        table.getCellRenderer (r, c);
    Component
        comp = renderer.getTableCellRendererComponent (
            table, table.getValueAt (r, c),
            false, false, r, c);

    height = comp.getMaximumSize ().height;
    maxh = height > maxh ? height : maxh;
}
return maxh;
}
}

```

例 19-18 列出了图 19-28 所示应用程序的完整代码。

例 19-18 计算和设置行高

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class Test extends JFrame {
    Object [] columnNames = {"Name", "Cost/Lb.", "Picture"};
    Object [] [] rowData = {
        {"cake", "$ 1.29", new ImageIcon ("cake.gif") },
        {"pear", "$ 1.29", new ImageIcon ("pear.gif") },
        {"pineapple", "$ 1.29", new ImageIcon ("pineapple.gif") },
        {"apple", "$ 1.29", new ImageIcon ("apple.gif") },
        {"bread", "$ 1.29", new ImageIcon ("bread.gif") },
    };

    class RowSizingModel extends DefaultTableModel {
        public RowSizingModel (Object [] [] data,
                               Object [] colNames) {
            super (data, colNames);
        }

        public Class getColumnClass (int c) {
            if (c == 2) return ImageIcon.class;
            else return super.getColumnClass (c);
        }
    }

    JTable table = new JTable (new RowSizingModel (rowData,
                                                    columnNames));

    public Test () {
        table.setRowHeight (getMaxRowHeight ());
        getContentPane ().add (new JScrollPane (table),
                                BorderLayout.CENTER);
    }

    public int getMaxRowHeight () {

```

```
int columnCount = table.getColumnCount (), h = 0, maxh = 0;

for (int i=0; i < columnCount; ++i) {
    TableColumn column =
        table.getColumnModel ().getColumn (i);

    h = getMaxRowHeightForColumn (column);
    maxh = h > maxh? h: maxh;
}

return maxh;
}

public int getMaxRowHeightForColumn (TableColumn column) {
    int height = 0, maxh = 0, c = column.getModelIndex ();

    for (int r=0; r < table.getRowCount (); ++r) {
        TableCellRenderer renderer =
            table.getCellRenderer (r, c);

        Component
            comp = renderer.getTableCellRendererComponent (
                table, table.getValueAt (r, c),
                false, false, r, c);

        height = comp.getMaximumSize ().height;
        maxh = height > maxh ? height : maxh;
    }

    return maxh;
}

public static void main (String args []) {
    GJApp.launch (
        new Test (), "Sizing Rows", 300, 300, 450, 300);
}
}
```

19.7.2 绘制行

Swing 表格的绘制器和编辑器在列的基础上被分配，然而有时在行或单元中进行绘制或编辑也是必要的。图 19-29 所示应用程序实现了一个基于单元的绘制器，同样在逻辑上可以对编辑进行扩展。

图 19-29 所示的表格的每一列都配置了一个绘制器，将偶数行单元绘制成蓝色；将奇数行单元绘制成橙色。

这个应用程序用 AbstractTableModel 的一个简单的扩展来实例化一个表格，而且这个应用程序的构造方法将奇数列的单元绘制器设置成 RowRenderer 的一个实例。

```
public class Test extends JFrame {

    //instantiate table with simple model...

    public Test () {
        TableColumn column;
        int columnCount = table.getColumnCount
        ();
```

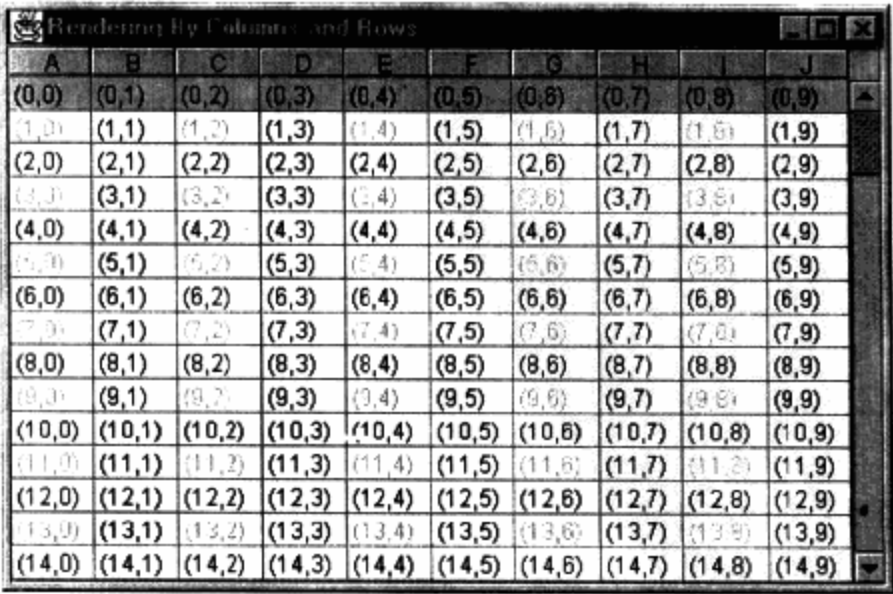


图 19-29 利用行进行绘制

```

for (int i=0; i < columnCount; ++i) {
    column = table.getColumn (table.getColumnName (i));
    if (i % 2 == 0)
        column.setCellRenderer (new RowRenderer ());
}
getContentPane ().add (new JScrollPane (table),
                        BorderLayout.CENTER);
}
...
}

```

RowRenderer 类重载 `getTableCellRendererComponent ()` 来设置行的前景色。随后从 `DefaultTableCellRenderer` 的 `getTableCellRendererComponent` 方法中获得绘制器组件。

```

class RowRenderer extends DefaultTableCellRenderer {
    public Component getTableCellRendererComponent (JTable table,
                                                    Object value, boolean isSelected,
                                                    boolean hasFocus,
                                                    int row, int column) {
        if (row % 2 == 0) setForeground (Color.blue);
        else              setForeground (Color.orange);
        return super.getTableCellRendererComponent (table,
                                                    value, isSelected, hasFocus,
                                                    row, column);
    }
}

```

例 19-19 列出了图 19-29 所示应用程序的完整代码。

例 19-19 利用行和列进行绘制

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;

public class Test extends JFrame {
    JTable table = new JTable (
        new AbstractTableModel () {
            int rows = 100, cols = 10;

            public int getRowCount () { return rows; }
            public int getColumnCount () { return cols; }

            public Object getValueAt (int row, int col) {
                return " (" + Integer.toString (row) + ", " +
                    Integer.toString (col) + ")";
            }
        }
    );

    public Test () {
        TableColumn column;
        int columnCount = table.getColumnCount ();
        for (int i=0; i < columnCount; ++i) {
            column = table.getColumn (table.getColumnName (i));
            if (i % 2 == 0)
                column.setCellRenderer (new RowRenderer ());
        }
    }
}

```

```

    }
    getContentPane().add(new JScrollPane(table),
        BorderLayout.CENTER);
}
public static void main (String args []) {
    GJApp.launch (
        new Test (), "Rendering By Columns and Rows",
        300, 300, 450, 300);
}

class RowRenderer extends DefaultTableCellRenderer {
    public Component getTableCellRendererComponent (JTable table,
        Object value, boolean isSelected,
        boolean hasFocus,
        int row, int column) {
        if (row % 2 == 0) setForeground (Color.blue);
        else setForeground (Color.orange);

        return super.getTableCellRendererComponent (table,
            value, isSelected, hasFocus,
            row, column);
    }
}
```

19.8 表格装饰器

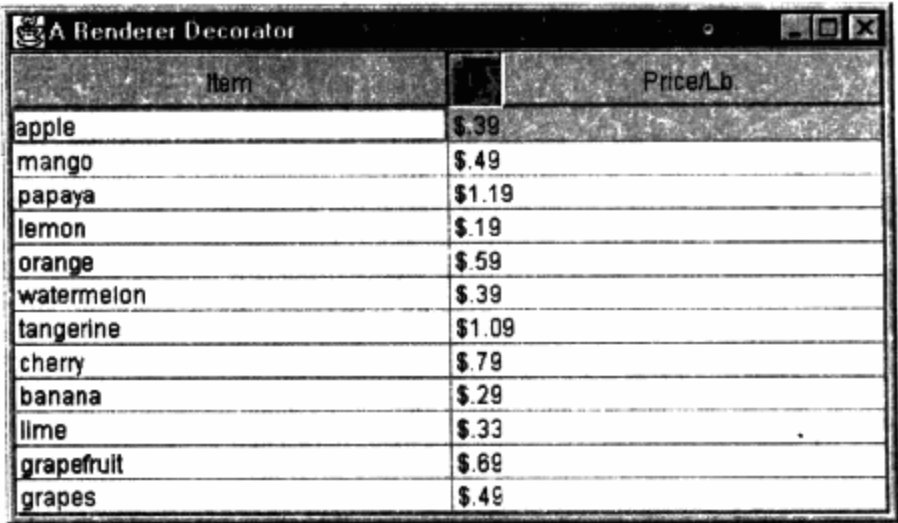
Java 语言中，对象经常使用继承来添加其功能。例如，由一个定制绘制器将一个图标添加到表格的列头部上，这个定制绘制器扩展 TableCellRenderer 并根据列头部的值来显示一个图标。

继承不允许将功能添加到运行中的单独对象中。例如，利用上述的定制绘制器，图标不能被添加到其他列头部上，只有配备有定制绘制器的那些列才能显示图标。

利用装饰器设计模式可以增加正在运行的对象的功能。装饰器将其功能交给一个封装的对象。装饰器实现了它们的封装对象的接口，所以装饰器能够代替该封装对象。装饰器把请求发送给它们的封装对象并执行附加的操作。○

装饰器最好用例子来说明。图 19-30 所示的表格的 Price/Lb 列配备了一个绘制装饰器，这个装饰器把一个图标添加到列头部中。

图 19-30 的应用程序创建一个具有简单模型的表格。这个应用程序的构造方法获得对 Price/Lb 列的一个引用，并随后获得对这个列的头部绘制器的一个引用。



Item	Price/Lb
apple	\$.39
mango	\$.49
papaya	\$1.19
lemon	\$.19
orange	\$.59
watermelon	\$.39
tangerine	\$1.09
cherry	\$.79
banana	\$.29
lime	\$.33
grapefruit	\$.69
grapes	\$.49

图 19-30 一个绘制装饰器

○ java.io.FilterInputStream 是装饰器模式的一个例子。

把这个列的头部设置成 `RendererDecorator` 的一个实例，这个实例是用这个列的初始头部绘制器来构造的。

```
public class Test extends JFrame {
    //create table with simple model

    public Test () {
        TableColumn column = table.getColumn ("Price/Lb.");
        TableCellRenderer renderer = column.getHeaderRenderer ();
        column.setHeaderRenderer (new RendererDecorator (renderer));
        getContentPane ().add (new JScrollPane (table),
                                BorderLayout.CENTER);
    }
    ...
}
```

`RendererDecorator` 类实现 `TableCellRenderer` 接口，所以它可以作为一个绘制器来传送，因此将它传送给 `TableColumn.setHeaderRenderer()`。`RendererDecorator` 类还维护一个对表格单元绘制器的引用，该引用被传送给 `RendererDecorator` 构造方法。

```
...
class RendererDecorator implements TableCellRenderer {
    TableCellRenderer realRenderer;
    JPanel panel;
    JLabel iconLabel = new JLabel (new ImageIcon ("money.gif"));

    public RendererDecorator (TableCellRenderer r) {
        realRenderer = r;
        iconLabel.setBorder (BorderFactory.createEtchedBorder ());
    }
    ...
}
```

当从 `Price/Lb.` 列的头部绘制器获得一个组件时，就调用这个装饰器的方法 `getTableCellRendererComponent`。装饰器获得绘制器的组件后将它放入一个具有标签的面板中进行修饰，该标签显示图标，随后这个面板作为绘制器组件返回。

```
...
public Component getTableCellRendererComponent (
    JTable table, Object value,
    boolean isSelected, boolean hasFocus,
    int row, int col) {
    Component c = realRenderer.getTableCellRendererComponent (
        table, value, isSelected,
        hasFocus, row, col);
    embellishComponent (c);
    return panel;
}

private void embellishComponent (Component c) {
    if (panel == null) {
        panel = new JPanel ();
        panel.setLayout (new BorderLayout ());
        panel.add (c, BorderLayout.CENTER);
        panel.add (iconLabel, BorderLayout.WEST);
    }
}
```

使用绘制装饰器的优点是可以通过 `RendererDecorator` 的一个实例，任何表格单元绘制器都能用一个图标来装饰。

例 19-20 列出图 19-30 所示应用程序的完整代码。

例 19-20 一个绘制装饰器

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class Test extends JFrame {
    JTable table = new JTable (new Object [] [] {
        {"apple", "$ .39"}, {"mango", "$ .49"},
        {"papaya", "$ 1.19"}, {"lemon", "$ .19"},
        {"orange", "$ .59"}, {"watermelon", "$ .39"},
        {"tangerine", "$ 1.09"}, {"cherry", "$ .79"},
        {"banana", "$ .29"}, {"lime", "$ .33"},
        {"grapefruit", "$ .69"}, {"grapes", "$ .49"},
    },
        new Object [] { "Item", "Price/Lb." });

    public Test () {
        TableColumn column = table.getColumnModel ("Price/Lb.");
        TableCellRenderer renderer = column.getHeaderRenderer ();
        column.setHeaderRenderer (new RendererDecorator (renderer));
        getContentPane ().add (new JScrollPane (table),
                                BorderLayout.CENTER);
    }

    public static void main (String args []) {
        GJApp.launch (
            new Test (), "A Renderer Decorator", 300, 300, 450, 182);
    }
}

class RendererDecorator implements TableCellRenderer {
    TableCellRenderer realRenderer;
    JPanel panel;
    JLabel iconLabel = new JLabel (new ImageIcon ("money.gif"));

    public RendererDecorator (TableCellRenderer r) {
        realRenderer = r;
        iconLabel.setBorder (BorderFactory.createEtchedBorder ());
    }

    public Component getTableCellRendererComponent (
        JTable table, Object value,
        boolean isSelected, boolean hasFocus,
        int row, int col) {
        Component c = realRenderer.getTableCellRendererComponent (
```

```
        table, value, isSelected,
        hasFocus, row, col);
    embellishComponent (c);
    return panel;
}

private void embellishComponent (Component c) {
    if (panel == null) {
        panel = new JPanel ();

        panel.setLayout (new BorderLayout ());
        panel.add (c, BorderLayout.CENTER);
        panel.add (iconLabel, BorderLayout.WEST);
    }
}
```

装饰器排序

装饰器可以用于表格列的排序。使用装饰器模式的排序允许不修改表格模型而对任何表格列进行排序。图 19-31 所示的应用程序包含一个配备了 SortDecorator 实例的表格。

图 19-31 中的上图显示了应用程序的初始的状态。中图显示了 Item 列排序后的表格，下图显示了 Price/Lb. 列排序后的表格。在列头部上进行鼠标点击就开始排序，由 SortDecorator 来执行排序。

图 19-32 示出了 SortDecorator 类的类图。
SortDecorator 类扩展 AbstractTableModel 类并

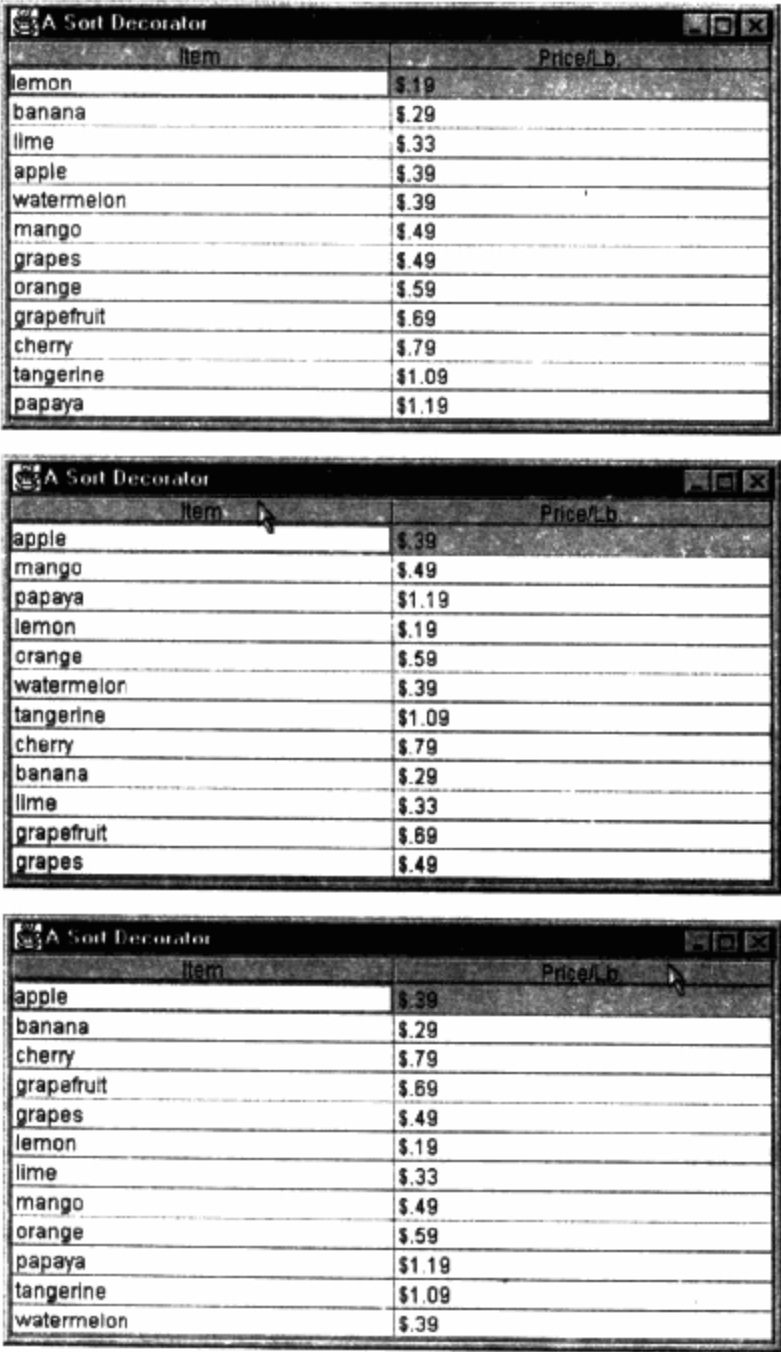


图 19-31 利用装饰器来进行表格排序

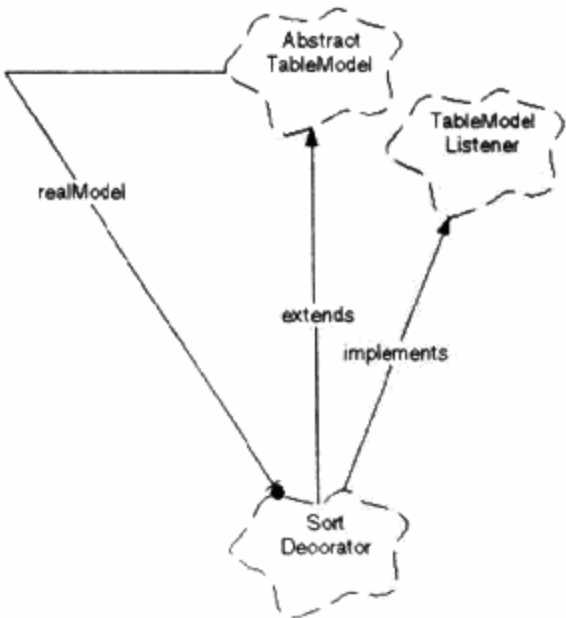


图 19-32 SortDecorator 类的类图

交给一个封装的表格模型。利用对其封装表格模型的监听，SortDecorator 类还实现了 TableModelListener 接口。

图 19-31 所示的应用程序利用定制模型创建了 JTable 的一个实例。这个应用程序的构造方法创建了 SortDecorator 的一个实例，该实例是用一个对表格初始模型的引用来创建的。随后，这个表格的模型设置成排序装饰器。

```
public class Test extends JFrame {
    //create table with model

    public Test () {
        final SortDecorator decorator =
            new SortDecorator (table.getModel ());

        table.setModel (decorator);
        ...
    }
}
```

添加到表格头部中的鼠标监听器响应鼠标的点击以对该列进行排序。

```
...
JTableHeader hdr = (JTableHeader) table.getTableHeader ();
hdr.addMouseListener (new MouseAdapter () {
    public void mouseClicked (MouseEvent e) {
        TableColumnModel tcm = table.getColumnModel ();
        int vc = tcm.getColumnIndexAtX (e.getX ());
        int mc = table.convertColumnIndexToModel (vc);

        decorator.sort (mc);
    }
});
getContentPane ().add (new JScrollPane (table),
                        BorderLayout.CENTER);
...
}
```

排序装饰器维护一个对表格实模型和代表已排序行索引的 integer 数组的引用。

SortDecorator 构造方法保存对实模型的引用并作为一个表格模型监听器将它自己加到实模型中。构造方法随后调用一个分配 indexes 数组的 private allocate 方法。

实模型的变化由 SortDecorator.tableChanged 方法处理，这个方法重新分配 indexes 数组并初始化这个数组中的整数（从 0 到 indexes.length-1）。

```
class SortDecorator extends AbstractTableModel,
    implements TableModelListener {
    private TableModel realModel;
    private int indexes [];

    public SortDecorator (TableModel model) {
        if (model == null)
            throw new IllegalArgumentException (
                "null models are not allowed");
        this.realModel = model;

        realModel.addTableModelListener (this);
        allocate ();
    }

    public void tableChanged (TableModelEvent e) {
```

```

        allocate ();
    }
    private void allocate () {
        indexes = new int [getRowCount ()];
        for (int i=0; i < indexes.length; ++i) {
            indexes [i] = i;
        }
    }
    ...

```

排序由 SortDecorator.sort 方法实现,这个方法重新安排存储在 indexes 数组中的索引。采用简单的排序算法,比较从单元值的 toString 方法返回的字符串并按顺序进行排序。注意,排序方法激发一个表格结构变化事件,参见本章 Swing 提示“AvstractTableModel 扩展必须激发相应的事件”。

```

    ...
    public void sort (int column) {
        int rowCount = getRowCount ();
        for (int i=0; i < rowCount; i++) {
            for (int j = i+1; j < rowCount; j++) {
                if (compare (indexes [i], indexes [j], column) < 0) {
                    swap (i, j);
                }
            }
        }
        fireTableStructureChanged ();
    }
    public void swap (int i, int j) {
        int tmp = indexes [i];
        indexes [i] = indexes [j];
        indexes [j] = tmp;
    }
    public int compare (int i, int j, int column) {
        Object io = realModel.getValueAt (i, column);
        Object jo = realModel.getValueAt (j, column);

        int c = jo.toString ().compareTo (io.toString ());
        return (c < 0) ? -1 : ( (c > 0) ? 1 : 0);
    }
    ...

```

SortDecorator 的 getValueAt 方法和 setValueAt 方法被重载以便存储和获取值 (用已排序的索引数组)。

```

    ...
    public Object getValueAt (int row, int column) {
        return realModel.getValueAt (indexes [row], column);
    }
    public void setValueAt (Object aValue, int row, int column) {
        realModel.setValueAt (aValue, indexes [row], column);
    }
    ...

```

最后, SortDecorator 通过把工作交给实模型实现了 TableModel 接口定义的其他方法。

```

    ...
    // TableModel pass-through methods follow ...

```

```

public int getRowCount () {
    return realModel.getRowCount ();
}
public int getColumnCount () {
    return realModel.getColumnCount ();
}
public String getColumnName (int columnIndex) {
    return realModel.getColumnName (columnIndex);
}
public Class getColumnClass (int columnIndex) {
    return realModel.getColumnClass (columnIndex);
}
public boolean isCellEditable (int rowIndex, int columnIndex) {
    return realModel.isCellEditable (rowIndex, columnIndex);
}
public void addTableModelListener (TableModelListener l) {
    realModel.addTableModelListener (l);
}
public void removeTableModelListener (TableModelListener l) {
    realModel.removeTableModelListener (l);
}
}

```

例 19-21 列出了图 19-31 所示应用程序的完整代码。

例 19-21 利用装饰器排序

```

import java.awt. * ;
import java.awt.event. * ;
import java.util. * ;
import javax.swing. * ;
import javax.swing.event. * ;
import javax.swing.table. * ;

public class Test extends JFrame {
    JTable table = new JTable (new Object [] [] {
        {"apple", "$ .39"}, {"mango", "$ .49"},
        {"papaya", "$ 1.19"}, {"lemon", "$ .19"},
        {"orange", "$ .59"}, {"watermelon", "$ .39"},
        {"tangerine", "$ 1.09"}, {"cherry", "$ .79"},
        {"banana", "$ .29"}, {"lime", "$ .33"},
        {"grapefruit", "$ .69"}, {"grapes", "$ .49"},
    },
        new Object [] { "Item", "Price/Lb." });

    public Test () {
        final SortDecorator decorator =
            new SortDecorator (table.getModel ());

        table.setModel (decorator);

        JTableHeader hdr = (JTableHeader) table.getTableHeader ();
        hdr.addMouseListener (new MouseAdapter () {
            public void mouseClicked (MouseEvent e) {
                TableColumnModel tcm = table.getColumnModel ();
                int vc = tcm.getColumnIndexAtX (e.getX ());
                int mc = table.convertColumnIndexToModel (vc);
            }
        });
    }
}

```

```

        decorator.sort (mc);
    }
    });
    getContentPane ().add (new JScrollPane (table),
                                BorderLayout.CENTER);
}
public static void main (String args []) {
    GJApp.launch (
        new Test (), "A Sort Decorator", 300, 300, 450, 250);
}

class SortDecorator implements TableModel, TableModelListener {
    private TableModel realModel;
    private int indexes [];

    public SortDecorator (TableModel model) {
        if (model == null)
            throw new IllegalArgumentException (
                "null models are not allowed");
        this.realModel = model;

        realModel.addTableModelListener (this);
        allocate ();
    }

    public Object getValueAt (int row, int column) {
        return realModel.getValueAt (indexes [row], column);
    }

    public void setValueAt (Object aValue, int row, int column) {
        realModel.setValueAt (aValue, indexes [row], column);
    }

    public void tableChanged (TableModelEvent e) {
        allocate ();
    }

    public void sort (int column) {
        int rowCount = getRowCount ();

        for (int i=0; i < rowCount; i++) {
            for (int j = i+1; j < rowCount; j++) {
                if (compare (indexes [i], indexes [j], column) < 0) {
                    swap (i, j);
                }
            }
        }
    }

    public void swap (int i, int j) {
        int tmp = indexes [i];
        indexes [i] = indexes [j];
        indexes [j] = tmp;
    }

    public int compare (int i, int j, int column) {
        Object io = realModel.getValueAt (i, column);
        Object jo = realModel.getValueAt (j, column);

        int c = jo.toString ().compareTo (io.toString ());
        return (c < 0) ? -1 : ( (c > 0) ? 1 : 0);
    }
}

```

```
private void allocate () {
    indexes = new int [getRowCount ()];
    for (int i=0; i < indexes.length; ++i) {
        indexes [i] = i;
    }
}

// TableModel pass-through methods follow ...

public int getRowCount () {
    return realModel.getRowCount ();
}

public int getColumnCount () {
    return realModel.getColumnCount ();
}

public String getColumnName (int columnIndex) {
    return realModel.getColumnName (columnIndex);
}

public Class getColumnClass (int columnIndex) {
    return realModel.getColumnClass (columnIndex);
}

public boolean isCellEditable (int rowIndex, int columnIndex) {
    return realModel.isCellEditable (rowIndex, columnIndex);
}

public void addTableModelListener (TableModelListener l) {
    realModel.addTableModelListener (l);
}

public void removeTableModelListener (TableModelListener l) {
    realModel.removeTableModelListener (l);
}
```

19.9 表格头部

表格头部是显示在表格的滚动窗格视口中的组件。如果表格不包含在滚动窗格中，那么表格的头部将是不可见的，如图 19-2 所示。

尽管表格头部是组件，但它们并不绘制自身。表格头部由表格列提供的绘制器进行绘制。有关列绘制器的更多信息，请参见“表格列”。

19.9.1 JTableHeader

表格头部由 JTableHeader 组件实现。JTableHeader 是两



图 19-33 JtableHeader 的类图

个不存在于 Swing 包中的其中一个 Swing 组件。[⊖] JTableHeader 可以在 swing.Table 包中找到，因为它是 Swing 开发人员直接使用的组件，是 Swing 表格的一个更细致的实现。

图 19-33 示出了 JTableHeader 的类图。

JTableHeader 扩展 JComponent 并实现 Accessible 和 TableColumnModelListener 接口。当修改列属性时，JTableHeader 的实例监听它们的列模型以便重新绘制和重新调整大小。

JTableHeader 维护对它的表格和表格列模型的一个引用。JTableHeader 还维护 TableColumn 引用以便跟踪正在调整大小或拖动的列。JTableHeader 还跟踪列被拖动的距离（作为一个 integer 值）。

JTableHeader 负责维护列是否能重新排序或重新调整大小的属性。JTableHeader 也跟踪拖动的列是否被实时地更新。

19.9.2 列头部绘制器和头部工具提示

表格头部不得不实现多行头部和头部工具提示的处理。两者的处理见图 19-34 所示的应用程序，这个应用程序实现了一个多行头部并为头部和中间的初始列设置了工具提示。

图 19-34 所示的图片显示了多行头部对表格宽度变化的反应（通过更新这个表格所显示的文本行数）。

注意 由于一个 JTextArea 错误，图 19-34 所示的多行头部当应用程序的窗口重新调整大小时未正确显示。

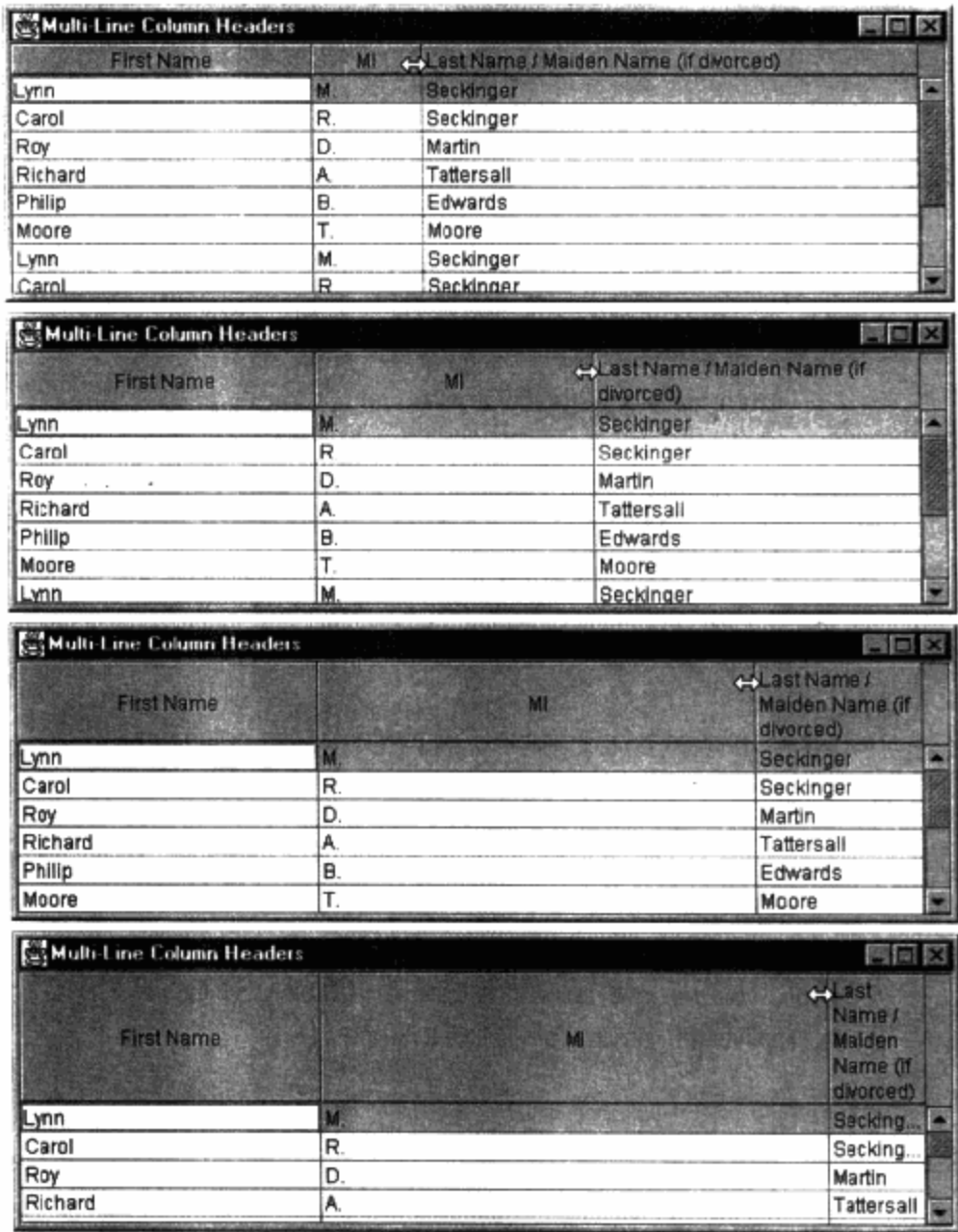


图 19-34 一个表格的多行列头部

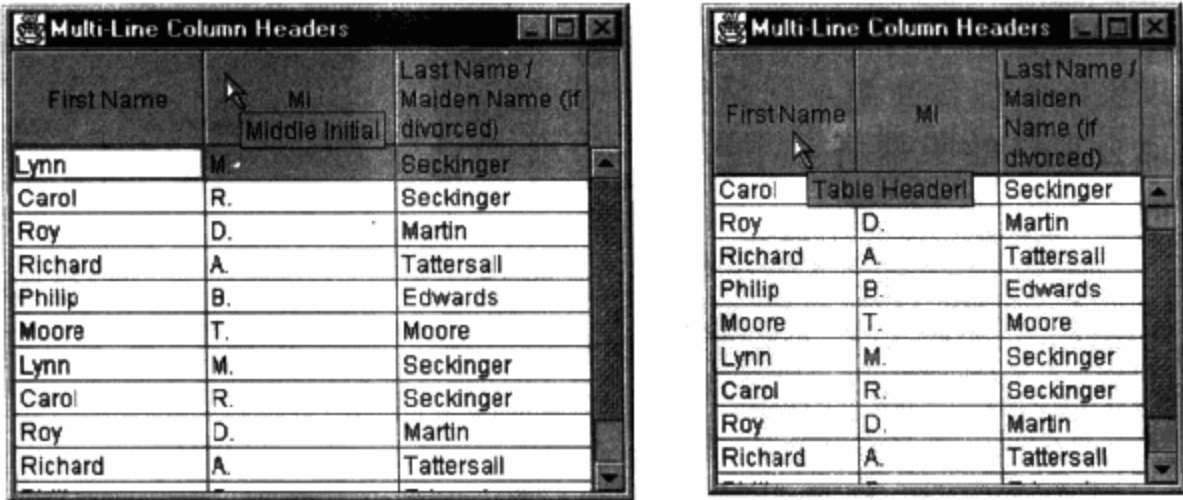


图 19-35 表格列头部工具提示

⊖ 另一个组件是来自 swing.text 包的 JTextComponent。

再重新调整窗口大小就可以解决这个问题。如图 19-34 所示，如果列重新调整大小，头部就像广告那样显示。

图 19-35 显示了 First Name 列中的表格头部的工具提示和这个列头部绘制器的工具提示。

这个应用程序的构造方法创建 MultilineHeaderRenderer 的一个实例，指定这个实例为 Last Name 列的头部绘制器。下面讨论 MultilineHeaderRenderer 类。

这个构造方法还获得对 MI 列的头部绘制器组件的一个引用。如果这个组件是 JComponent 的一个实例（缺省值），那么它的工具提示文本被设置。另外，为表格头部设置了工具提示。

头部绘制器指定的工具提示重载一个表格头部的工具提示。例如，显示在 MI 列的工具提示文本就是头部绘制器的工具提示文本。

```
public class Test extends JFrame {
    String longTitle = "Last Name / Maiden Name (if divorced)";
    MultilineHeaderRenderer multilineRenderer =
        new MultilineHeaderRenderer (longTitle);

    JTable table = new JTable (
        new Object [] [] {
            { "Lynn", "M.", "Seckinger" },
            //other row data omitted
        },
        new Object [] { "First Name", "MI", longTitle });

    public Test () {
        TableColumn middleColumn = table.getColumnModel () .getColumn ("MI"),
            lastColumn = table.getColumnModel () .getColumn (longTitle);

        lastColumn.setHeaderRenderer (multilineRenderer);

        TableCellRenderer hdrRenderer =
            middleColumn.getHeaderRenderer ();

        Component hdrComponent =
            hdrRenderer.getTableCellRendererComponent (table,
                "MI", false, false, 0, 0);

        if (hdrComponent instanceof JComponent) {
            JComponent c = (JComponent) hdrComponent;
            c.setToolTipText ("Middle Initial");
        }

        table.getTableHeader () .setToolTipText ("Table Header!");

        getContentPane () .add (
            new JScrollPane (table), BorderLayout.CENTER);
    }
    ...
}
```

MultilineHeaderRenderer 实现 TableCellRenderer 接口并返回一个包含 MultilineHeader 实例的滚动窗格将在下面讨论。

将多行头部包裹在滚动窗格中的原理讨论如下。

```
...
class MultilineHeaderRenderer implements TableCellRenderer {
    MultilineHeader mlh;
    JScrollPane scrollPane;
```



```

public MultilineHeaderRenderer (String title) {
    mlh = new MultilineHeader (title);
    scrollPane = new JScrollPane (mlh);

    scrollPane.setHorizontalScrollBarPolicy (
        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

    scrollPane.setVerticalScrollBarPolicy (
        JScrollPane.VERTICAL_SCROLLBAR_NEVER);

    scrollPane.setBorder (null);
}

public Component getTableCellRendererComponent (JTable table,
    Object value,
    boolean isSelected,
    boolean hasFocus,
    int row, int col) {
    mlh.setText ((String) value);
    return scrollPane;
}
}
...

```

MultilineHeader 类扩展 JTextArea，因为文本域提供了在字边界上使其文本换行的能力。MultilineHeader 有换行的功能，设置换行类型并把文本域的增亮性设置为 null，且不允许进行编辑。

MultilineHeader 重载 updateUI() 以使文本域看起来和操作起来像一个表格头部。无论何时安装界面样式，都调用 updateUI 方法，且由 MultilineHeader 重载的方法安装颜色、字体和一个边框，供表格头部的界面样式所使用。

```

...
class MultilineHeader extends JTextArea {
    public MultilineHeader (String s) {
        super (s);
    }

    public void updateUI () {
        super.updateUI ();

        // turn on wrapping and disable editing and highlighting

        setLineWrap (true);
        setWrapStyleWord (true);
        setHighlighter (null);
        setEditable (false);

        // make the text area look like a table header

        LookAndFeel.installColorsAndFont (this,
            "TableHeader.background",
            "TableHeader.foreground",
            "TableHeader.font");

        LookAndFeel.installBorder (this, "TableHeader.cellBorder");
    }
}

```

由 MultilineHeaderRenderer 返回的多行头部包裹在滚动窗格中，因为放在滚动窗格中的文本域的宽度跟踪滚动窗格的宽度。因此，多行头部的宽度总是与列头部的宽度相同，当列宽度变化时，就会导致文本显示行数的变化。有关 Scrollable 接口以及如何利用 JTextArea 类来实现这

个接口的更多信息，请参见“Scrollable 接口”。

例 19-22 列出了图 19-35 所示应用程序的完整代码。

例 19-22 一个多行表格列头部和头部工具提示

```
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.table.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    String longTitle = "Last Name / Maiden Name (if divorced)";
    MultilineHeaderRenderer multilineRenderer =
        new MultilineHeaderRenderer (longTitle);

    JTable table = new JTable (
        new Object [ ] [ ] {
            { "Lynn", "M.", "Seckinger" },
            { "Carol", "R.", "Seckinger" },
            { "Roy", "D.", "Martin" },
            { "Richard", "A.", "Tattersall" },
            { "Philip", "B.", "Edwards" },
            { "Moore", "T.", "Moore" },
            // shorten scrollbar grip with these ...
            { "Lynn", "M.", "Seckinger" },
            { "Carol", "R.", "Seckinger" },
            { "Roy", "D.", "Martin" },
            { "Richard", "A.", "Tattersall" },
            { "Philip", "B.", "Edwards" },
            { "Moore", "T.", "Moore" },
        },
        new Object [ ] { "First Name", "MI", longTitle } );

    public Test () {
        TableColumn middleColumn = table.getColumnModel () .getColumn ( "MI" ),
            lastColumn = table.getColumnModel () .getColumn ( longTitle );

        lastColumn.setHeaderRenderer ( multilineRenderer );

        TableCellRenderer hdrRenderer =
            middleColumn.getHeaderRenderer ();

        Component hdrComponent =
            hdrRenderer.getTableCellRendererComponent ( table, "MI", false, false, 0, 0 );

        if ( hdrComponent instanceof JComponent ) {
            JComponent c = ( JComponent ) hdrComponent;
            c.setToolTipText ( "Middle Initial" );
        }

        table.getTableHeader () .setToolTipText ( "Table Header!" );

        getContentPane () .add (
            new JScrollPane ( table ), BorderLayout.CENTER );
    }

    public static void main ( String args [ ] ) {
        GJApp.launch ( new Test () );
    }
}
```

```

        "Multi-Line Column Headers", 300, 300, 300, 250);
    }
}

class MultilineHeaderRenderer implements TableCellRenderer {
    MultilineHeader mlh;
    JScrollPane scrollPane;

    public MultilineHeaderRenderer (String title) {
        mlh = new MultilineHeader (title);
        scrollPane = new JScrollPane (mlh);

        scrollPane.setHorizontalScrollBarPolicy (
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

        scrollPane.setVerticalScrollBarPolicy (
            JScrollPane.VERTICAL_SCROLLBAR_NEVER);

        scrollPane.setBorder (null);
    }

    public Component getTableCellRendererComponent (JTable table,
        Object value,
        boolean isSelected,
        boolean hasFocus,
        int row, int col) {
        mlh.setText ((String) value);
        return scrollPane;
    }
}

class MultilineHeader extends JTextArea {
    public MultilineHeader (String s) {
        super (s);
    }

    public void updateUI () {
        super.updateUI ();

        // turn on wrapping and disable editing and highlighting

        setLineWrap (true);
        setWrapStyleWord (true);
        setHighlighter (null);
        setEditable (false);

        // make the text area look like a table header

        LookAndFeel.installColorsAndFont (this, "TableHeader.background", "TableHeader.foreground", "TableHeader.font");
        LookAndFeel.installBorder (this, "TableHeader.cellBorder");
    }
}

```

组件总结 19-1 JTable

模型: TableModel
 UI 代表: javax.swing.basic.BasicTableUI
 绘制器: TableCellRenderer
 编辑器: TableCellEditor
 激发的事件: PropertyChangeEvents, TableModelEvents

TableModelEvents, ListSelectionEvents

替代：——

类图：见图 19-36

JTable 类扩展 JComponent 并实现了许多接口。与其他 Swing 组件一样，JTable 实现 Accessible 和 Scrollable 接口。由 JTable 类实现的监听器接口能监听、响应其模型和编辑器激发的事件。

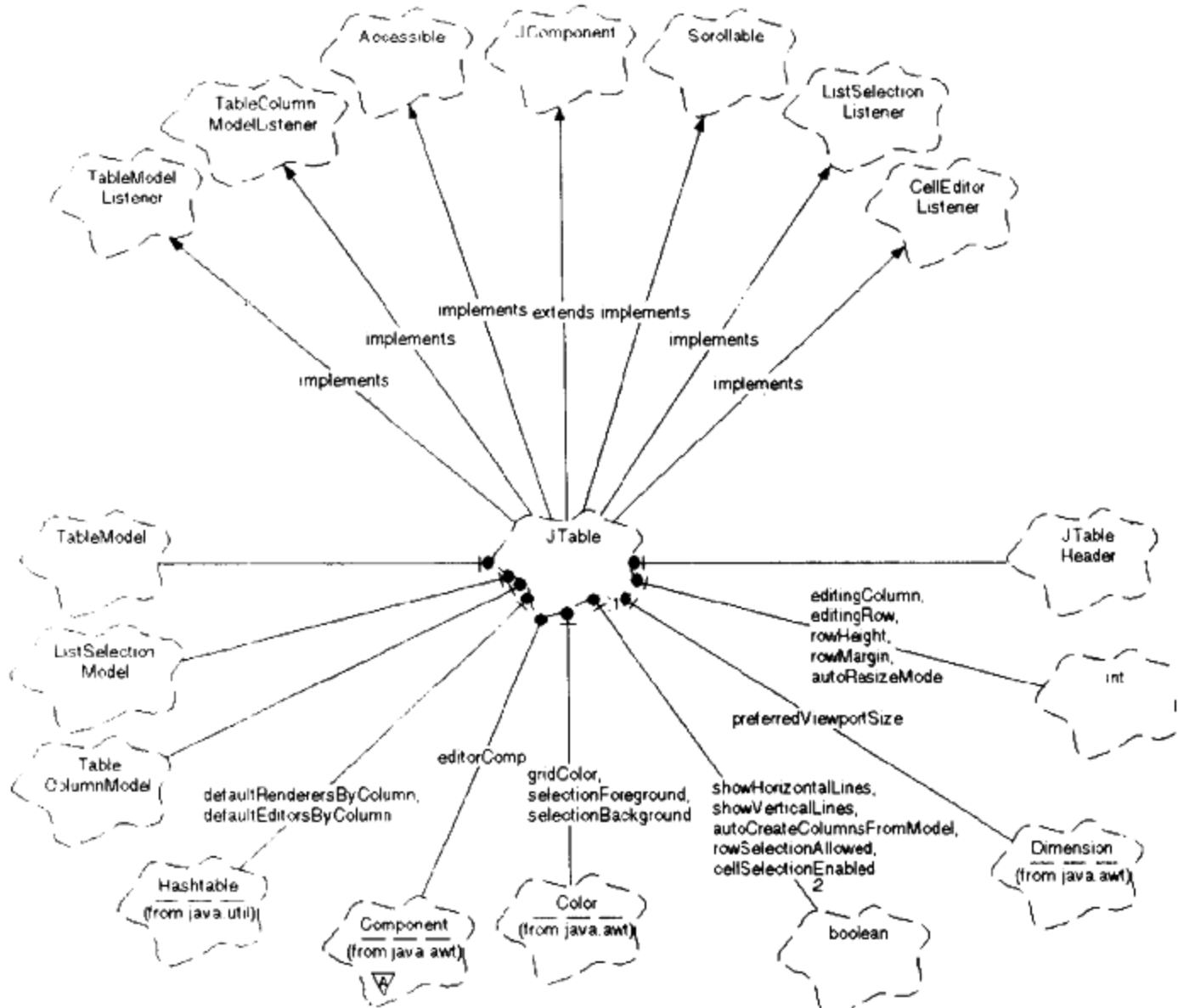


图 19-36 JTable 的类图

JTable 维护对其三种模型以及一个 TableHeader 实例和一个编辑器组件的 protected 引用。哈希表维护缺省的绘制器和编辑器。另外，JTable 还维护了许多用于跟踪其属性的 boolean 和 integer 值。有关 JTable 属性的详细内容，请参见 19.9.3 节“JTable 属性”。

19.9.3 JTable 属性

表 19-7 列出了由 JTable 类维护的属性。

表 19-7 JTable 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
autoCreateColumnsFromModel	boolean	S	SG	true
autoResizeMode	int	S	SG	见下面的讨论
cellEditable	boolean	S	G	——
cellSelectionEnabled	boolean	S	SG	false
columnModel	TableColumnModel	B	SG	DefaultTable-

(续)

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
				ColumnModel
columnSelectionAllowed	boolean	S	SG	false
defaultEditor	TableCellEditor	S	SG	——
defaultRenderer	TableCellRenderer	S	SG	见下面的讨论
editing	boolean	S	SG	false
editingColumn	int	S	SG	-1
editingRow	int	S	SG	-1
editorComponent	Component	S	G	——
gridColor	Color	S	SG	见下面的讨论
intercellSpacing	Dimension	TCM	SG	1
model	TableModel	B	SG	DefaultTableModel
rowHeight	int	S	SG	16
rowMargin	int	S	SG	1
rowCount	int	——	G	——
rowSelectionAllowed	boolean	S	SG	true
selectionBackground	Color	B	SG	UIM
selectionForeground	Color	B	SG	UIM
selectionMode	int	S	S	MULTIPLE _ SELECTION _ INTERVAL
SelectionMode	ListSelection- Model	S	SG	DefaultList- SelectionMode
ShowGrid	boolean	S	SG	true
ShowHorizontalLines	boolean	S	SG	true
ShwoVerticalLines	boolean	S	SG	true
TableHeader	JTableHeader	S	SG	JTableHeader

① B = 关联的 (激发 `PropertyChangeEvent`) / C = 受约束的 / I = 索引的 /
S = 简单的 / Ch = 激发 `ChangeEvent` / TCM = 表格列模型激发 `TableColumnModelEvent` / LS = 选取模型激发 `ListSelectionEvent` / TM
= 表格模型激发 `TableModelEvent`
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ I&F = 与界面样式有关 / UIM = `UIManager` 设置缺省值

autoCreateColumnsFromModel——决定表格是否从其模型中创建缺省的列。如果属性为 true，就用 `setModel` 方法设置表格模型，这个设置将删除已存在的列而创建新的列。属性缺省值是 true。

autoResizeMode——五种调整大小模式之一，能用 `setAutoResizeMode` 方法设置。有关调整大小模式信息参见“列调整大小模式”。

cellEditable——决定单元值是否可编辑。方法 `JTable.isCellEditable()` 让其模型决定一个单元是否可编辑。注意，该属性表示单元值是否是可编辑的，不必是单元本身。单元编辑器既能使单元值是可编辑的，也能指定一个单元是不可编辑的。

cellSelectionEnabled——指定行和列的选取是否能够同时存在。如果单元选取有效，则可重载行和列的选取模式。

columnModel——一个实现 TableModel 接口的表格模型。

columnSelectionAllowed——决定列是否能选取的列模型。

defaultEditor——一个特定类对象的缺省编辑器。JTable 给特定的数据类型提供了许多缺省编辑器，如果需要，就可以重载这些编辑器。

defaultRenderer——一个特定类对象的缺省绘制器，JTable 给特定的数据类型提供许多缺省绘制器，如果需要，就可以重载这些绘制器。

editing——指定一个表格单元当前是否正在编辑的一个只读属性。

editingColumn——如果表格单元正在编辑，那么 editingColumn 属性表明单元的列正在被编辑。

editingRow——如果表格单元正在编辑，那么 editingRow 属性表明单元的行正在被编辑。

editorComponent——如果表格单元正在编辑，那么 editorComponent 属性表明组件正用于编辑单元。

gridColor——用于画网格线的颜色。调用 UIManager.getColor (“Table. gridColor”) 方法从 UI 管理员中获得表格网格线的颜色。

intercellSpacing——表示单元间水平和垂直边距的尺寸。垂直边距利用表格的列模型获得，而水平边距由表格的 rowMargin 属性得到。

model——一个表格的模型，是实现 TableModel 接口的一个对象。

rowHeight——表格行高与所给的表格同高，rowHeight 属性决定一个表格总行的高度。行高必须大于 0——如果其值小于或等于 0，那么 JTable.setRowHeight () 将给出 IllegalArgumentException 的信息。

rowMargin——行间垂直边距。

rowCount——表格中显示的行数。

rowSelectionAllowed——决定表格行是否能被选取。属性值缺省为 true。

selectionBackground——选中单元的背景颜色。

selectionForeground——选中单元的前景颜色。

selectionMode——JTable 类支持的五种选取模式之一。有关选取模式的信息参见“表格选取”。

selectionModel——一个表格的选取模型，缺省为 DefaultListSelectionModel 的实例。

showGrid——决定是否显示水平和垂直行，设置这个属性来设置 showHorizontalLines 和 showVerticalLines 属性。属性值缺省为 showVerticalLines。

showHorizontalLines——决定是否显示水平网格线。缺省属性值是 true。

showVerticalLines——决定是否显示垂直网格线。缺省属性值是 true。

tableHeader——表格的头部，是 JTableHeader 类的一个实例。

19.9.4 表格事件

当表格的三个模型中的任意一个模型发生变化时，表格将激发相应的事件，见表 19-8。

表 19-8 表格事件

事件	被 ... 激发	激发当 ...
TableModelEvent	表格模型	单元值变化；表格结构变化； 行被插入/删除/更新；单元被更新
TableColumnModelEvent	表格列模型	列被添加/删除/移动；列边距发生变化；列选取发生变化
ListSelectionEvent	表格选取模型	行选取发生变化
ChangeEvent	DefaultCellEditor	编辑被停止/取消

JTable 类不提供任何添加监听器的方法。监听器必须直接加到一个表格的三种模型或一个单元编辑器上。表格模型的讨论参见 19.2 节“表格模型”。^①

19.9.5 表格模型事件

当一个表格模型发生变化时，将激发表格模型事件。由 TableModelListener 来处理表格模型事件，接口总结 19-6 定义了 TableModelListener。

接口总结 19-6 TableModelListener

```
public abstract void tableChanged (TableModelEvent)
```

TableModelListener.tableChanged 方法的参数是 TableModelEvent 的一个实例，类总结 19-6 总结了 TableModelEvent

类总结 19-6 TableModelEvent

1. 常量

```
public static final int DELETE
public static final int INSERT
public static final int UPDATE

public static final int ALL_COLUMNS
public static final int HEADER_ROW
```

上面列举的常量指定了事件的类型。前三个常量由 TableModelEvent.getType () 方法返回。

ALL_COLUMNS 和 HEADER_ROW 常量分别指定变化的列和变化的首行。ALL_COLUMNS 指定表格中每列可能已经变化。同样，HEADER_ROW 指明头部行已经变化，即列的名字、类型和顺序可能已经变化。

2. 构造方法

```
public TableModelEvent (TableModel)
public TableModelEvent (TableModel, int row)
public TableModelEvent (TableModel, int firstRow, int lastRow)
public TableModelEvent (TableModel, int firstRow, int lastRow, int column)
public TableModelEvent (TableModel, int firstRow, int lastRow, int column, int type)
```

TableModelEvent 类提供许多构造方法来指定不同类型的事件。一般情况下，开发人员不会

① 监听器登记方法在以后的 Swing 版本可能会增加到 JTable 类中。

启动表格模型事件，因为事件激发方法由 AbstractTableModel 实现。

3. 方法

```
public int getColumn ()
public int getFirstRow ()
public int getLastRow ()
public int getType ()
```

TableModelEvent 类提供上述方法以获得事件的有关信息。

图 19-37 所示应用程序含有一个表格，其模型配备了一个 TableModelListener 实例。图 19-37 上图显示了一个表格单元正在编辑。单元被编辑后，激发 TableModelEvent，通过显示消息对话框来处理这个激发的事件。

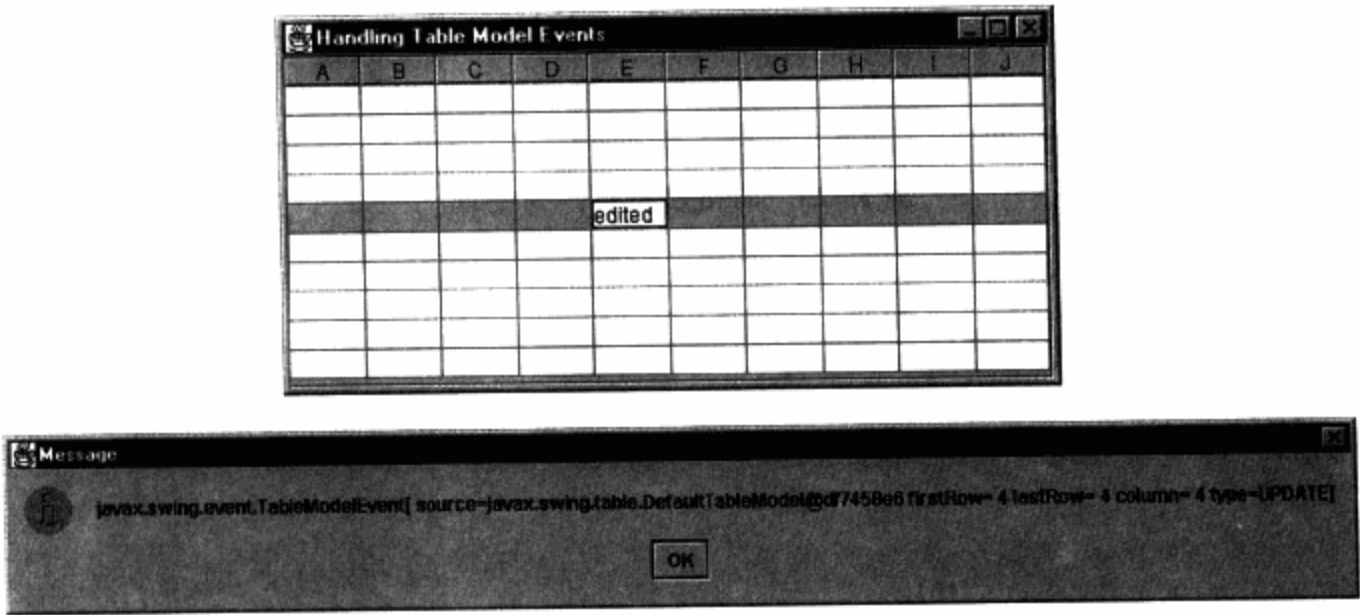


图 19-37 处理表格模型事件

这个应用程序创建了具有简单模型的表格并将 TableModelListener 添加到表格模型中。

```
public class Test extends JFrame {
    JTable table = new JTable (10, 10);

    public Test () {
        Container contentPane = getContentPane ();

        table.getModel ().addTableModelListener (
            new TableModelListener () {
                public void tableChanged (TableModelEvent e) {
                    int firstRow = e.getFirstRow (),
                        column = e.getColumn ();

                    String properties = " source=" + e.getSource () +
                        " firstRow=" +
                            (firstRow == TableModelEvent.HEADER_ROW ?
                                "HEADER_ROW" :
                                Integer.toString (firstRow)) +
                        " lastRow=" + e.getLastRow () +
                        " column=" +
                            (firstRow == TableModelEvent.ALL_COLUMNS ?
                                "ALL_COLUMNS" :
                                Integer.toString (column));

                    String typeString = new String ();
```

```

        int type = e.getType ();
        switch ( type ) {
            case TableModelEvent.DELETE:
                typeString = "DELETE"; break;
            case TableModelEvent.INSERT:
                typeString = "INSERT"; break;
            case TableModelEvent.UPDATE:
                typeString = "UPDATE"; break;
        }
        properties += " type=" + typeString;
        JOptionPane.showMessageDialog (Test.this,
            e.getClass ().getName () +
            " [" + properties + "]");
    }
}

```

例 19-23 列出了图 19-37 所示应用程序的完整代码。

例 19-23 表格模型事件的处理

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    JTable table = new JTable (10, 10);

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.add (new JScrollPane (table),
            BorderLayout.CENTER);

        table.getModel ().addTableModelListener (
            new TableModelListener () {
                public void tableChanged (TableModelEvent e) {
                    int firstRow = e.getFirstRow (),
                        column = e.getColumn ();

                    String properties = " source=" + e.getSource () +
                        " firstRow=" +
                            (firstRow == TableModelEvent.HEADER_ROW ?
                                "HEADER_ROW" :
                                Integer.toString (firstRow)) +
                        " lastRow=" + e.getLastRow () +
                        " column=" +
                            (firstRow == TableModelEvent.ALL_COLUMNS ?
                                "ALL_COLUMNS" :
                                Integer.toString (column));

                    String typeString = new String ();
                    int type = e.getType ();
                    switch (type) {
                        case TableModelEvent.DELETE:

```

```
        typeString = "DELETE"; break;
    case TableModelEvent.INSERT:
        typeString = "INSERT"; break;
    case TableModelEvent.UPDATE:
        typeString = "UPDATE"; break;
    }
    properties += " type=" + typeString;
    JOptionPane.showMessageDialog (Test.this,
        e.getClass ().getName () +
        " [" + properties + "]");
    }
    }

    public static void main (String args []) {
        GJApp. launch (new Test (),
            "Handling Table Model Events", 300, 300, 450, 220);
    }
}
```

19.9.6 TableColumnModel 事件

正常环境下，把 TableColumnModel 及其相关的事件看作是 JTable 类的一个实现。

TableColumnModel 事件由 TableColumnModelListener 实例来处理，并利用 TableColumnModel.add TableColumnModelListener 方法向列模型登记。

接口总结 19-7 总结了 TableColumnModelListener 接口

接口总结 19-7 TableColumnModelListener

```
public abstract void columnAdded (TableColumnModelEvent)
public abstract void columnMarginChanged (ChangeEvent)
public abstract void columnMoved (TableColumnModelEvent)
public abstract void columnRemoved (TableColumnModelEvent)
public abstract void columnSelectionChanged (ListSelectionEvent)
```

TableColumnModelListener 能检测出列的添加、移动或从表格列模型中删除以及列边距和选取的变化。

TableColumnModelListener 接口定义的所有方法，除 columnMarginChanged ()和 columnSelectionChanged ()外，其参数是 TableColumnModelEvent。类总结 19-7 总结了 TableColumnModelEvent 类。

类总结 19-7 TableColumnModelEvent

1. 构造方法

```
public TableColumnModelEvent (TableColumnModel model, int from,
    int to)
```

TableColumnModelEvent 类只提供了一个构造方法，这个构造方法的参数是 TableColumnModel 的一个引用和被事件影响的列索引。

2. 方法

```
public int getFromIndex ()
```

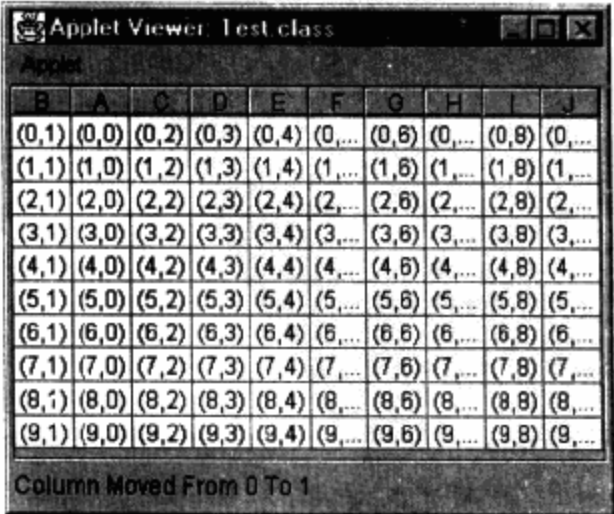


图 19-38 处理列模型事件

```
public int getToIndex ()
```

以上上列举的方法返回被事件影响的列索引。例如图 19-38 所示小应用程序添加一个 `TableColumnModelListener` 到表格列模型中。当移动一列时，监听器在小应用程序的状态区显示了由于移动事件而变化的列索引。

例 19-24 列出了图 19-38 所示小应用程序的完整代码。

例 19-24 列模型事件的处理

```
import javax.swing.*.*;
import javax.swing.event.*;
import javax.swing.table.*;
import java.awt.*.*;
import java.awt.event.*;

public class Test extends JApplet {
    JTable table = new JTable (
        new AbstractTableModel () {
            int rows = 10, cols = 10;

            public int getRowCount () { return rows; }
            public int getColumnCount () { return cols; }

            public Object getValueAt (int row, int col) {
                return " (" + Integer.toString (row) + ", " +
                    Integer.toString (col) + ")";
            }
        }
    );

    public void init () {
        Container contentPane = getContentPane ();
        contentPane.add (new JScrollPane (table),
            BorderLayout.CENTER);

        table.getColumnModel ().addColumnModelListener (
            new TableColumnModelListener () {
                public void columnAdded (TableColumnModelEvent e) {}
                public void columnMarginChanged (ChangeEvent e) {}
                public void columnRemoved (TableColumnModelEvent e) {}
                public void columnSelectionChanged (
                    ListSelectionEvent e) {}

                public void columnMoved (TableColumnModelEvent e) {
                    String s = "Column Moved From " +
                        e.getFromIndex () + " To " +
                        e.getToIndex ();

                    showStatus (s);
                }
            }
        );
    }
}
```

实现 `TableColumnModelListener` 接口的类必须提供代表无兴趣事件的方法的空实现，因为 Swing 没有为表格列模型监听器提供一个相应的类。

以上列举的监听器对列移动事件作出反应，它从这个事件中获得 From 和 To 索引，随后监听器更新这个小应用程序的状态区。

19.9.7 列表选取事件

列表选取事件由实现 ListSelectionListener 接口的对象来处理，其总结参见 17.3.2 节。

图 19-3 所示的应用程序包含一个表格，其选取模型配备了列表选取监听器。当选取事件发生时，监听器显示有关选取事件的信息。

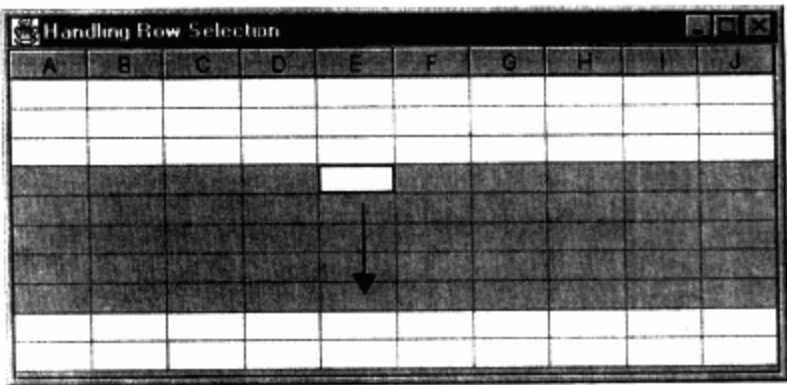


图 19-39 处理行选取事件

图 19-39 描述了鼠标从单元 (3, 4) 拖动到单元 (7, 4) 的情况。当把光标拖动过单元时将激发调整事件。当释放鼠标按键时，激发非调整事件。

```
Selection Model: javax.swing.event.ListSelectionEvent [
Source = javax.swing.DefaultListSelectionModel 2048245 = {0}
FirstIndex = 0 lastIndex = 0 isAdjusting = false]

Selection Model adjusting ...
Selection Model adjusting ...
Selection Model adjusting ...
Selection Model adjusting ...
Selection Model adjusting ...

Selection Model: javax.swing.event.ListSelectionEvent [
Source = javax.swing.DefaultListSelectionModel 2048245 = {3, 4, 5, 6, 7}
FirstIndex = 0 lastIndex = 7 isAdjusting = false]
```

这个应用程序创建了一个简单模型的表格并将一个监听器加到表格的选取模型中。如果与事件相关的值正在调整，那么监听器显示字符串 “adjusting...”，未调整的事件由事件的 toString 方法来处理。

```
public class Test extends JFrame {
    JTable table = new JTable (10, 10);

    public Test () {
        //add table, wrapped in scrollpane, to content pane...

        table.getSelectionModel ().addListSelectionListener (
            new ListSelectionListener () {
                public void valueChanged (ListSelectionEvent e) {
                    if (e.getValueIsAdjusting ()) {
                        System.out.println ("Selection Model " +
                            "adjusting ...");
                    }
                    else {
                        System.out.println ("Selection Model:" +
                            e.toString ());
                    }
                }
            }
        );
    }
}
```

例 19-25 列出了图 19-39 所示应用程序的完整代码。

例 19-25 行选取事件的处理

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    JTable table = new JTable (10, 10);

    public Test () {
        Container contentPane = getContentPane ();
        contentPane.add (new JScrollPane (table),
            BorderLayout.CENTER);

        table.getSelectionModel ().addListSelectionListener (
            new ListSelectionListener () {
                public void valueChanged (ListSelectionEvent e) {
                    if (e.getValueIsAdjusting ()) {
                        System.out.println ("Selection Model " +
                            "adjusting ...");
                    }
                    else {
                        System.out.println ("Selection Model:" +
                            e.toString ());
                    }
                }
            }
        );
    }

    public static void main (String args []) {
        GJApp.launch (new Test (),
            "Handling Row Selection", 300, 300, 450, 220);
    }
}

```

19.9.8 JTable 类总结

类总结 19-8 列出了 JTable 的 public 和 protected 方法。

类总结 19-8 JTable

1. 构造方法

```

public JTable ()
public JTable (int numRows, int numColumns)
public JTable (Object [] [], Object [])
public JTable (Vector, Vector)
public JTable (TableModel)
public JTable (TableModel, TableColumnModel)
public JTable (TableModel, TableColumnModel, ListSelectionModel)

```

JTable 类提供许多利用数据和模型建立表格的构造方法。

利用缺省值、无参数构造方法给表格模型，表格列模型以及列表选取模型创建了一个 JTable 实例。还可以用 JTable 构造方法来构造表格，这个 JTable 构造方法以这个表格中的行数和列数为参数。

上面列举的最后五个构造方法其传送参数都是格式化的数据，前两个构造方法其传送参数是 Object 数组和矢量并配置 AbstractTableModel 的简单扩展。最后三个构造方法其传送参数是表格模型、表格列模型和列表选取模型。如果三种模型都为 null，那么模型由以下列举的 protected JTable 方法创建。

2. 方法

(1) 初始化

```
protected TableColumnModel createDefaultColumnModel ()
protected TableModel createDefaultDataModel ()
protected ListSelectionModel createDefaultSelectionModel ()
protected void createDefaultEditors ()
protected void createDefaultRenderers ()
protected JTableHeader createDefaultTableHeader ()
protected void initializeLocalVars ()
```

以上列举的方法创建缺省模型、编辑器、绘制器和一个表格头部。创建的缺省模型是 DefaultTableModel、DefaultTableColumnModel 和 DefaultListSelectionModel 的实例。缺省表格头部是 JTableHeader 的一个实例。

以上列举方法创建的绘制器和编辑器缺省对应于一定的对象类型，参见表 19-4。

方法 initializeLocalVars 将 JTable 属性初始化成缺省值，参见表 19-7。

(2) 表格和滚动窗格

```
public static JScrollPane createScrollPaneForTable (JTable)
public void addNotify ()
protected void configureEnclosingScrollPane ()
```

在 Swing 早期，表格必需被添加到特殊配置的滚动窗格中，该滚动窗格是调用方法 static createScrollPaneForTable 创建的，现已取消。

随后，重载 addNotify（当一个表格加到一个容器中时调用该方法）以便消除 createScrollPaneForTable 方法的需求。JTable.addNotify 先调用方法 super.addNotify()，然后再调用方法 configureEnclosingScrollPane()。

方法 configureEnclosingScrollPane 检查表格是否包含在一个滚动窗格中。如果表格包含在一个滚动窗格中，那么表格的头部就被设置成滚动窗格的头部视图，在滚动窗格中能够进行后台存储，且设置滚动窗格为一个特殊的视感边框。

(3) TableModelListener 方法

```
public void tableChanged (TableModelEvent)
```

利用 tableChanged 方法，JTable 的实例可以响应模型的变化。依据变化类型采取适当动作，对表格重新绘制并调整大小。

(4) TableColumnModelListener 方法

```
public void columnAdded (TableColumnModelEvent)
public void columnMarginChanged (ChangeEvent)
public void columnMoved (TableColumnModelEvent)
public void columnRemoved (TableColumnModelEvent)
```



```
public void columnSelectionChanged (ListSelectionEvent)
```

以上列举的方法由 `TableColumnListener` 接口定义。`JTable` 实例作为监听器向其列模型登记, 且以上列举的方法响应表格列模型中激发的事件。

上面列举的几个方法检查当前是否正在进行编辑, 如果是这样, 则停止编辑并将编辑器从表格中删除, 方法随后依据变化类型重新绘制或调整表格大小。上面列举的最后一个方法响应列选取变化并重新绘制受影响的列。

(5) ListSelectionListener 方法

```
Public void valueChanged (ListSelectionEvent)
```

利用 `ListSelectionListener` 方法的实现, `JTable` 实例也可以监听它们的选取模型。`JTable.ValueChanged()` 重新绘制受选取变化影响的列。

(6) Scrollable 方法

```
public int getScrollableBlockIncrement (Rectangle visibleRect, int orientation, int direction)
public boolean getScrollableTracksViewportHeight ()
public boolean getScrollableTracksViewportWidth ()
public int getScrollableUnitIncrement (Rectangle, int, int)
public Dimension getPreferredScrollableViewportSize ()
```

以上列举的方法由 `Scrollable` 接口定义, 这个接口由 `JTable` 类实现。块和单元滚动增量的设置参见“块增量和单元增量”。

如果 `autoResizeMode` 属性未被设置成 `JTable.AUTO_RESIZE_OFF`, 那么 `JTable` 实例跟踪表格视口的宽度。缺省情况下, 表格不跟踪视口的高度。

(7) CellEditorListener 方法

```
public void editingCanceled (ChangeEvent)
public void editingStopped (ChangeEvent)
```

利用 `CellEditorListener` 接口的实现, `JTable` 可以监听单元编辑器。`editingStopped()` 和 `editingCanceled()` 从表格中删除编辑器, 且 `editingStopped()` 使用编辑器的值来设置正在编辑的单元值。有关表格编辑的信息参见“绘制和编辑”。

(8) 选取

```
public void addColumnSelectionInterval (int, int)
public int getSelectedColumn ()
public int getSelectedColumnCount ()
public int [] getSelectedColumns ()

public int getSelectedRow ()
public int getSelectedRowCount ()
public int [] getSelectedRows ()

public void addRowSelectionInterval (int, int)
public void clearSelection ()

public void removeColumnSelectionInterval (int, int)
public void removeRowSelectionInterval (int, int)

public void setSelectionBackground (Color)
public void setSelectionForeground (Color)

public void setSelectionMode (int)
public void setSelectionModel (ListSelectionModel)

public void setCellSelectionEnabled (boolean)
public void setColumnSelectionAllowed (boolean)
public void setColumnSelectionInterval (int, int)
```

```

public boolean getCellSelectionEnabled ()
public boolean getColumnSelectionAllowed ()
public void selectAll ()

```

表格支持选取行、列和单元。以上列举方法能设置选取模式、添加或删除行和列的间隔，并设置表格的选取模式。另外，JTable 实例维护前景和背景颜色的选取，并可以由以上列举的访问方法访问。方法 `selectAll` 选取表格中所有的行和列。

(9) 编辑表格单元

```

public TableCellEditor getCellEditor ()
public TableCellEditor getCellEditor (int row, int column)
public TableCellEditor getDefaultEditor (Class)
public void setDefaultEditor (Class, TableCellEditor)

public int getEditingColumn ()
public int getEditingRow ()

public boolean editCellAt (int row, int column)
public boolean editCellAt (int row, int column, EventObject)
public Component getEditorComponent ()
public boolean isEditing ()
public Component prepareEditor (TableCellEditor, int row, int column)

public void removeEditor ()
public void setEditingColumn (int)
public void setEditingRow (int)
public void setCellEditor (TableCellEditor)

```

上面列举的方法通过提供对当前编辑器的访问来支持表格编辑。编辑行和列的方法不应该由开发人员直接调用。

(10) 绘制表格单元

```

public TableCellRenderer getCellRenderer (int row, int column)
public TableCellRenderer getDefaultRenderer (Class)
public void setDefaultRenderer (Class, TableCellRenderer)
public Component prepareRenderer (TableCellRenderer, int row, int column)

```

上面列出的方法通过提供对缺省绘制器和特定表格单元绘制器的访问来支持表格单元的绘制。绘制期间，表格通过上面列出的方法来满足其全部的绘制要求。因此，可以通过任何方式（中央列的缺省实现除外）重载这些方法来提供绘制器。

(11) 实用方法

```

public void sizeColumnToFit (int)
public int convertColumnIndexToModel (int)
public int convertColumnIndexToView (int)
public void createDefaultColumnsFromModel ()
public int columnAtPoint (Point)
public int rowAtPoint (Point)
public void addColumn (TableColumn)
protected void resizeAndRepaint ()
public void moveColumn (int, int)
protected String paramString ()
public void removeColumn (TableColumn)
public void reshape (int x, int y, int w, int h)

```

以上列举的方法是执行多种功能的杂项实用方法，包括在适当位置返回响应的行或列并移动和删除表格列。

(12) 杂项属性

```

public TableColumn getColumn (Object)
public Class getColumnClass (int)
public int getColumnCount ()
public TableColumnModel getColumnModel ()
public String getColumnName (int)
public boolean getAutoCreateColumnsFromModel ()
public int getAutoResizeMode ()
public Color getGridColor ()
public Dimension getInterCellSpacing ()
public TableModel getModel ()
public int getRowCount ()
public int getRowHeight ()
public int getRowMargin ()
public boolean getRowSelectionAllowed ()
public Color getSelectionBackground ()
public Color getSelectionForeground ()
public ListSelectionModel getSelectionModel ()
public boolean getShowHorizontalLines ()
public boolean getShowVerticalLines ()
public JTableHeader getTableHeader ()
public String getToolTipText (MouseEvent)
public Object getValueAt (int, int)

public boolean isCellSelected (int, int)
public boolean isColumnSelected (int)
public boolean isManagingFocus ()
public boolean isRowSelected (int)

public void setAutoCreateColumnsFromModel (boolean)
public void setAutoResizeMode (int)
public void setColumnModel (TableColumnModel)
public void setGridColor (Color)
public void setInterCellSpacing (Dimension)
public void setModel (TableModel)
public void setRowHeight (int)
public void setRowMargin (int)
public void setSelectionAllowed (boolean)
public void setSelectionInterval (int, int)
public void setShowGrid (boolean)
public void setShowHorizontalLines (boolean)
public void setShowVerticalLines (boolean)
public void setTableHeader (JTableHeader)
public void setValueAt (Object, int, int)

```

上述方法是 JTable 杂项属性的访问方法。有关 JTable 属性的更多信息，请参见“JTable 属性”。

(13) 可访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext ()
public TabbedPaneUI getUI ()
public String getUIClassID ()
public void setUI (TabbedPaneUI)
public void updateUI ()

```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI

代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

19.9.9 AWT 兼容

AWT 不提供与 Swing 的 JTable 相类似的组件。

19.10 本章回顾

JTable 具有三种模式，一个头部组件和一个单元绘制器和编辑器，它是 Swing 中最复杂的组件，因而本章内容最多。使用表格并不难，但表格的许多不同的概念（表格列、头部、绘制器、编辑器等等）使学习起来非常困难。因此本章介绍了许多执行表格任务的技巧，这些任务如隐藏列（19.4.3 节的“隐藏列”）和利用行而不是列来进行绘制（19.7.2 节的“用行进行绘制”）等。

第 20 章 树

Swing 树使用人们所熟悉的文件夹和树叶图来显示分层的数据。应用最广泛的树组件^①无疑是 Windows Explorer，它包含一个用于导航目录的树组件。

与表格类似，树由许多类和接口组成，这些类和接口在它们自己的包——swing.tree 包中定义，swing 包中的 JTree 类代表树组件。

树由节点组成，节点可以是文件夹，也可以是树叶。文件夹可以有子节点，除根节点之外的所有节点都只有一个父节点。空的文件夹与树叶的不同之处就在于它允许有子节点。

图 20-1 显示的是 JTree 类的一个扩展，该扩展可用于导航目录和文件。文件夹和树叶由不同的图标表示，这些图标都是彼此独立的。在文件夹上双击，或单击文件夹的句柄，就可以展开或折叠文件夹，根节点句柄的可见性可以被设置。例如，图 20-1 所示的树的根节点就没有显示句柄。

除父节点和子节点外，树的节点还有一个用户对象^②。

用户对象是 Object 类型，因此它提供了一个将任意对象与节点相关联的办法。

树有一个简单的模型，每一个 JTree 实例都要维护对绘制器和编辑器的引用，这个绘制器和编辑器被树中所有的节点所使用。在表 20-1 中列出了 swing.tree 包中的主要类。

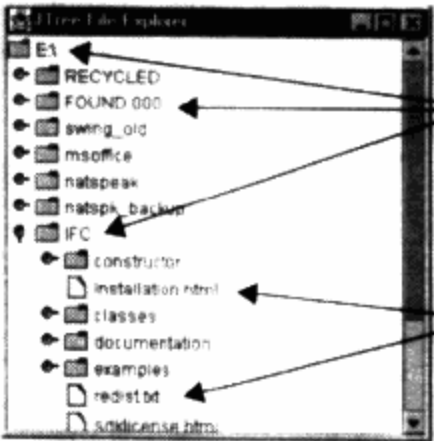


图 20-1 Jtree 结构

表 20-1 Swing.tree 包中的主要类

名称	实现
DefaultMutableTreeNode	一个具有一个父节点、(可能)许多子节点和一个用户对象的可变节点，为相关联的节点提供了访问方法。如果没有任何子节点，这个节点就是树叶
DefaultTreeModel	一个激发 TreeModel Events 事件的简单可变的模型。提供对子节点的访问方法，但不提供对父节点的访问方法
DefaultTreeCellEditor	绘制器和编辑器的包装器，它把一个“真正”的编辑器组件放在节点图标旁边
DefaultTreeCellRenderer	具有字体、颜色和图标访问方法的 JLabel 扩展，它提供图标的缺省值
TreePath	由一个节点到另一个节点的路径。路径中的节点存储在一个数组中。路径用于在选取内容之间进行通信

20.1 创建树

图 20-2 示出的小应用程序包含一个表格，这个表格是用 JTree 无参数构造方法创建的。

① 树组件又称为轮廓控件。
② 当使用 DefaultTreeModel 时，就会呈现一个用户对象。

图 20-2 中所示的小应用程序向应用程序的内容窗格中添加了一个树，这个树包裹在一个滚动窗格中。在例 20-1 中列出了这个小应用程序的代码。在缺省情况下显示一个树时，它的文件夹初始状态是展开的。图 20-2 中所示的树在程序开始运行时，它的文件夹都是展开的。

如果在构造时没有显式地指定模型或节点，则 JTree 的实例就以图 20-2 所示的节点来构造。

几乎所有的树都是以下面的方式来构造的：先创建一个根节点，然后建立分层结构

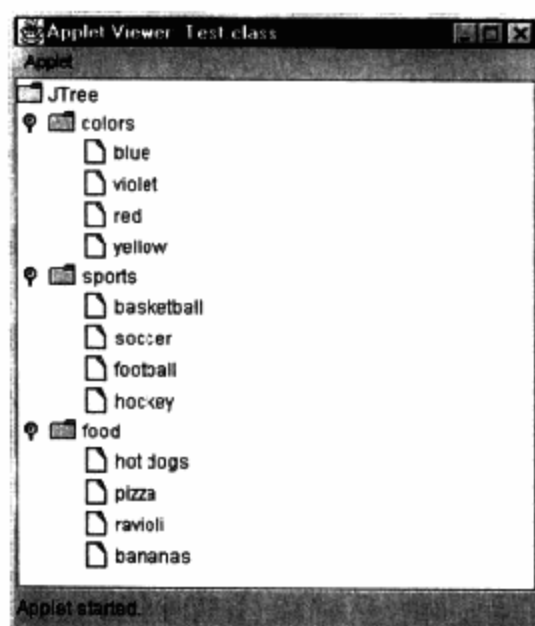


图 20-2 缺省的树节点

例 20-1 一个简单的树的例子

```
import javax.swing.*.*;

public class Test extends JApplet {
    public void init () {
        getContentPane ().add (new JScrollPane (new JTree ()));
    }
}
```

或创建一个树模型。例如，在创建图 20-2 中所示的节点的缺省分层结构时，JTree 缺省的构造方法就调用了 JTree.getDefaultTreeModel ()。

```
// From JTree.java:

protected static TreeModel getDefaultTreeModel () {
    DefaultMutableTreeNode root =
        new DefaultMutableTreeNode ("JTree");
    DefaultMutableTreeNode parent;

    parent = new DefaultMutableTreeNode ("colors");
    root.add (parent);

    parent.add (new DefaultMutableTreeNode ("blue"));
    parent.add (new DefaultMutableTreeNode ("violet"));
    parent.add (new DefaultMutableTreeNode ("red"));
    parent.add (new DefaultMutableTreeNode ("yellow"));

    parent = new DefaultMutableTreeNode ("sports");
    root.add (parent);

    parent.add (new DefaultMutableTreeNode ("basketball"));
    parent.add (new DefaultMutableTreeNode ("soccer"));
    parent.add (new DefaultMutableTreeNode ("football"));
    parent.add (new DefaultMutableTreeNode ("hockey"));

    ...

    return new DefaultTreeModel (root);
}
```

用户对象的带有字符串 JTree 的节点被实例化，而且最终指定为树模型的根节点，节点 colors 有四个子节点，且指定为根节点的唯一子节点。节点 sports 也有四个节点，而且也被添

加到根节点中。

JTree 类还提供了用 Object 数组、哈希表和矢量创建树的构造方法，如图 20-3 所示。

与构造分层节点相比，用数据来构造树，需要注意两点：

第一点，数据很少（几乎没有）用来构造树。通常，树都是用节点来配置的，图 20-2 所示的小应用程序即是一例。

第二点，由于对象添加到哈希表中的顺序和对象在哈希表中存放的顺序没有任何联系，因此，用哈希表创建的树的节点顺序是难以预料的。

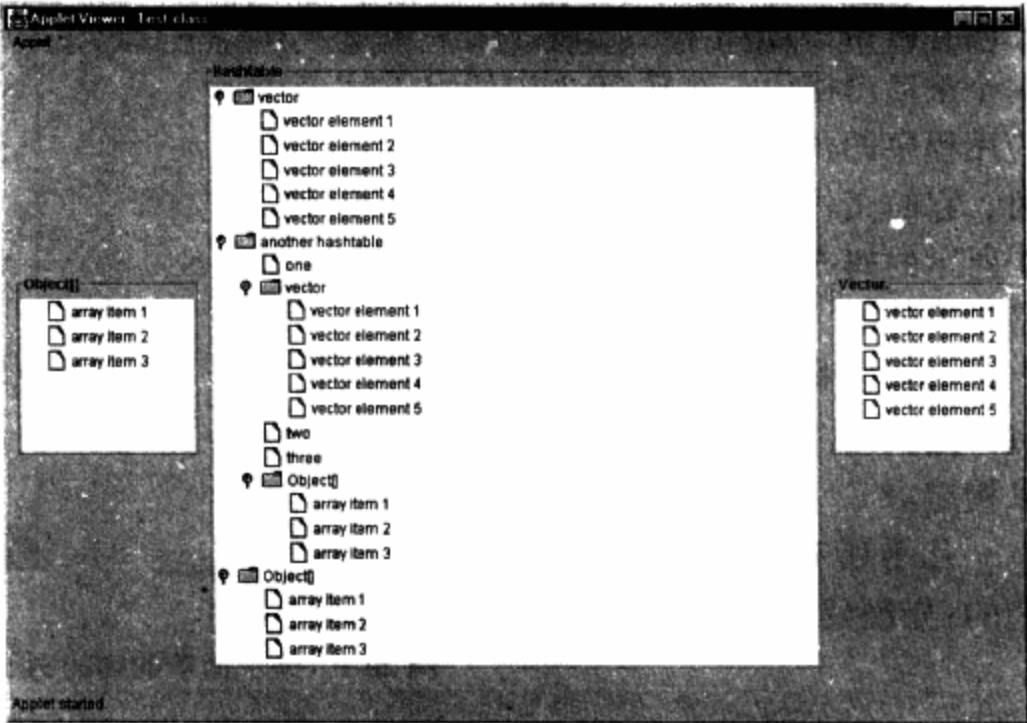


图 20-3 用对象、矢量和哈希表来创建树

例 20-2 列出了图 20-3 中所示小应用程序的完整代码。

例 20-2 用对象、矢量和哈希表来创建树

```
import javax.swing.* ;
import javax.swing.tree.TreePath;
import java.awt.* ;
import java.awt.event.* ;
import java.util.* ;

public class Test extends JApplet {
    Hashtable ht = new Hashtable (), ht2 = new Hashtable ();
    Vector vector = new Vector ();
    Object [] objs = new Object [] {
        "array item 1", "array item 2", "array item 3"
    };
    public void init () {
        Container contentPane = getContentPane ();

        vector.addElement ("vector element 1");
        vector.addElement ("vector element 2");
        vector.addElement ("vector element 3");
        vector.addElement ("vector element 4");
        vector.addElement ("vector element 5");

        ht.put ("another hashtable", ht2);
        ht.put ("vector", vector);
        ht.put ("Object []", objs);

        ht2.put ("Object []", objs);
        ht2.put ("vector", vector);
        ht2.put ("one", new Integer (1));
        ht2.put ("two", new Integer (2));
        ht2.put ("three", new Integer (3));

        // trees must be created after data is populated
        JTree hashTree = new JTree (ht);
```



```

JTree vectorTree = new JTree (vector);
JTree objectTree = new JTree (objs);
JScrollPane objPane = new JScrollPane (objectTree);
JScrollPane hashPane = new JScrollPane (hashTree);
JScrollPane vectorPane = new JScrollPane (vectorTree);
objPane.setPreferredSize (new Dimension (150, 150));
hashPane.setPreferredSize (new Dimension (500, 500));
vectorPane.setPreferredSize (new Dimension (150, 150));
objPane.setBorder (
    BorderFactory.createTitledBorder ("Object [ ]"));
hashPane.setBorder (
    BorderFactory.createTitledBorder ("Hashtable"));
vectorPane.setBorder (
    BorderFactory.createTitledBorder ("Vector"));
hashTree.expandPath (new TreePath (
    hashTree.getModel ().getRoot ());
contentPane.setLayout (new FlowLayout ());
contentPane.add (objPane);
contentPane.add (hashPane);
contentPane.add (vectorPane);

```

20.2 树节点

在 Swing 树中，树节点是关键的部分，如同列是表格的主干一样。树节点由 `TreeNode` 接口定义，`TreeNode` 接口被 `MutableTreeNode` 接口扩展，而 `MutableTreeNode` 接口又由 `Default Mutable TreeNode` 类来实现。

20.2.1 `TreeNode` 接口

`TreeNode` 接口定义了（固定）树节点的实质，接口总结 20-1 总结了树节点。

接口总结 20-1 `TreeNode`

```

public abstract Enumeration children ()
public abstract TreeNode getParent ()
public abstract TreeNode getChildAt (int)
public abstract int getChildCount ()
public abstract int getIndex (TreeNode)
public abstract boolean getAllowsChildren ()
public abstract boolean isLeaf ()

```

上面列出的前两组方法是对一个节点的父节点和子节点的访问方法。访问一个节点的子节点，可以通过枚举子节点的父节点来实现，也可以通过索引来访问子节点。另外，还定义了获取节点索引的方法和获取一个节点包含的子节点数目的方法。

上面列出的最后两个方法用来确定一个节点是文件夹，还是树叶。

开发人员很少直接实现 `TreeNode` 接口，这是因为 Swing 在 `DefaultMutableTreeNode` 类中提供

了 `TreeNode` 接口的一个常用的缺省实现。数目众多的树节点扩展了 `DefaultMutableTreeNode`。

20.2.2 MutableTreeNode 接口

`MutableTreeNode` 接口扩展 `TreeNode`，它除了定义指定用户对象的方法以外，还定义了修改一个节点的父节点和子节点的方法。接口总结 20-2 总结了 `MutableTreeNode` 接口。

接口总结 20-2 MutableTreeNode

扩展: `TreeNode`

```
public abstract void insert (MutableTreeNode child, int index)
public abstract void remove (int index)
public abstract void remove (MutableTreeNode child)
public abstract void removeFromParent ( )

public abstract void setParent (MutableTreeNode)
public abstract void setUserObject (Object)
```

上面列出的第一组方法用来插入和删除子节点，子节点可以通过索引或引用来删除。`removeFromParent` 方法用来将节点从父节点中删除，并更新父节点的子节点数目。

上面列出的最后两个方法用来设置一个节点的父节点和用户对象。需要注意的是，`MutableTreeNode` 继承了 `getParent` 方法，而没有继承 `getUserObject` 方法，这是一个疏漏，在以后发布的 Swing 中将予以更正。在实际应用中，因为没有 `getUserObject` 方法而造成的影响几乎为零，这是因为该方法已在 `DefaultMutableTreeNode` 类中得到了实现。

20.2.3 DefaultMutableTreeNode 类

在实际应用中，很少直接实现 `MutableTreeNode` 接口，这是因为 Swing 以 `DefaultMutableTreeNode` 类的形式提供了一个合理又强壮的 `MutableTreeNode` 接口的实现。

图 20-4 示出了 `DefaultMutableTreeNode` 类的类图。

`DefaultMutableTreeNode` 类实现 `MutableTreeNode` 接口，并维护对其父节点、用户对象和子节点的引用。通过维护对其父节点和子节点的一个引用，`DefaultMutableTreeNode` 类实现了组合设计样式^①，该样式允许对文件夹和树叶进行嵌套。AWT 的 `Component` 类和 `Container` 类也是组合设计样式的一个样例^②。

1. 使用 DefaultMutableTreeNode

树节点几乎总是 `DefaultMutableTreeNode` 类的实例或它的扩展。例如，由一个树创建的缺省节点就是 `DefaultMutableTreeNode` 的实例。

除了那些在 `TreeNode` 和 `MutableTreeNode` 接口中定义的方法外，`DefaultMutableTreeNode` 类还提供了许多其他的方法来访问相关的节点。例如，它提供了 `getFirstLeaf` 和 `getNextLeaf` 方法，前一种方法返回第一个树叶，第二种方法返回某个给定树叶的下一个兄弟节点。

`DefaultMutableTreeNode` 类还提供了一些方法，这些方法以深度优先或宽度优先的遍历方式来返回由一个节点派生的子节点的一个枚举，深度优先和宽度优先这两种遍历方式的差别见图 20-5。

例 20-3 的小应用程序显示树的节点，这些节点是通过对一个树的根节点调用 `Default-`

① 参见 Addison-Wesley 于 1995 年出版的由 Gamma、Helm、Johnson 和 Vlissides 合著的《Design Patterns》。

② 参见由 Sun Microsystems Press, Prentice Hall 于 1998 年出版的 Geary、David 合著的《Java 2 图形设计，卷 I：AWT》。

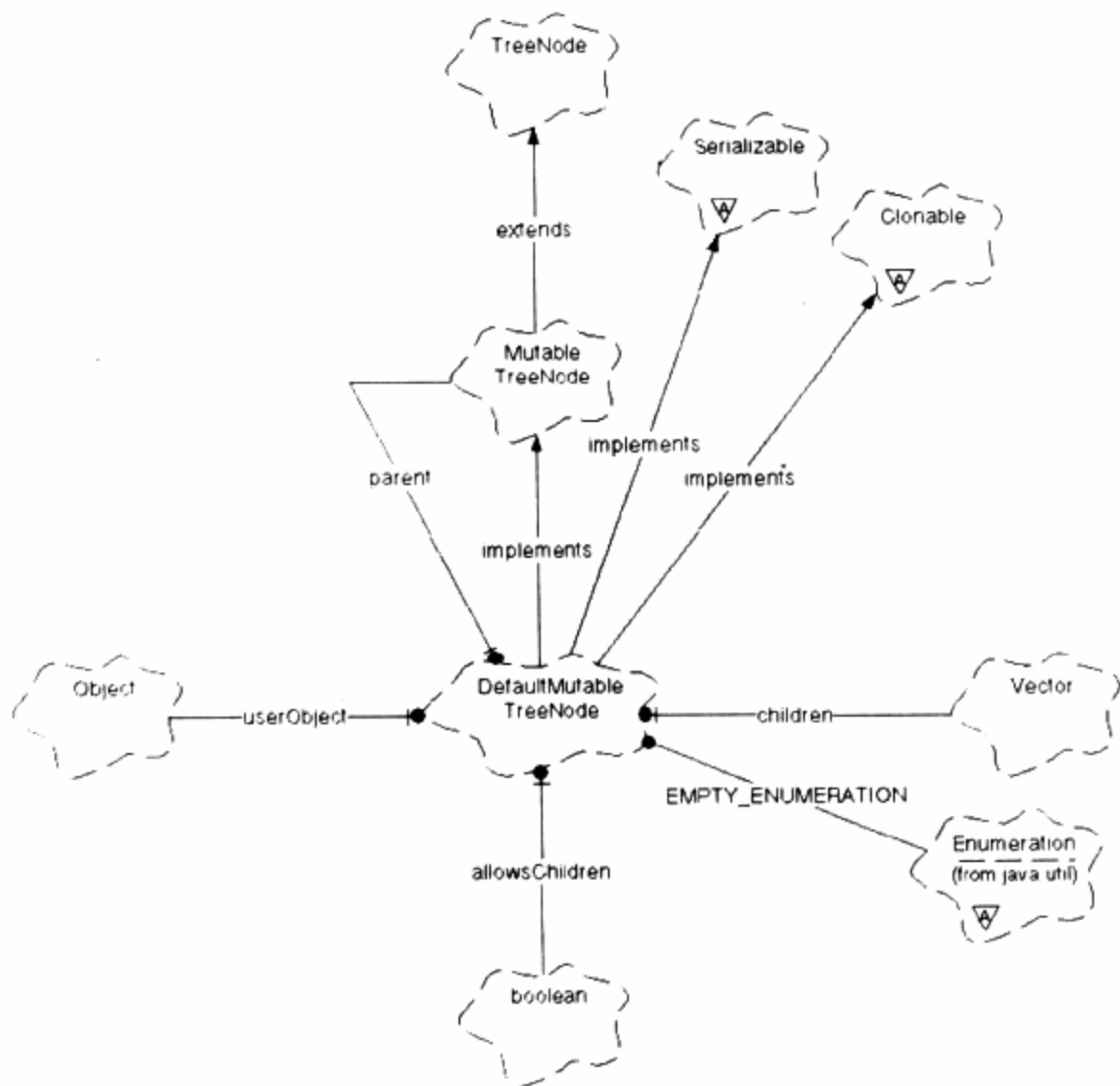


图 20-4 DefaultMutableTreeNode 的类图

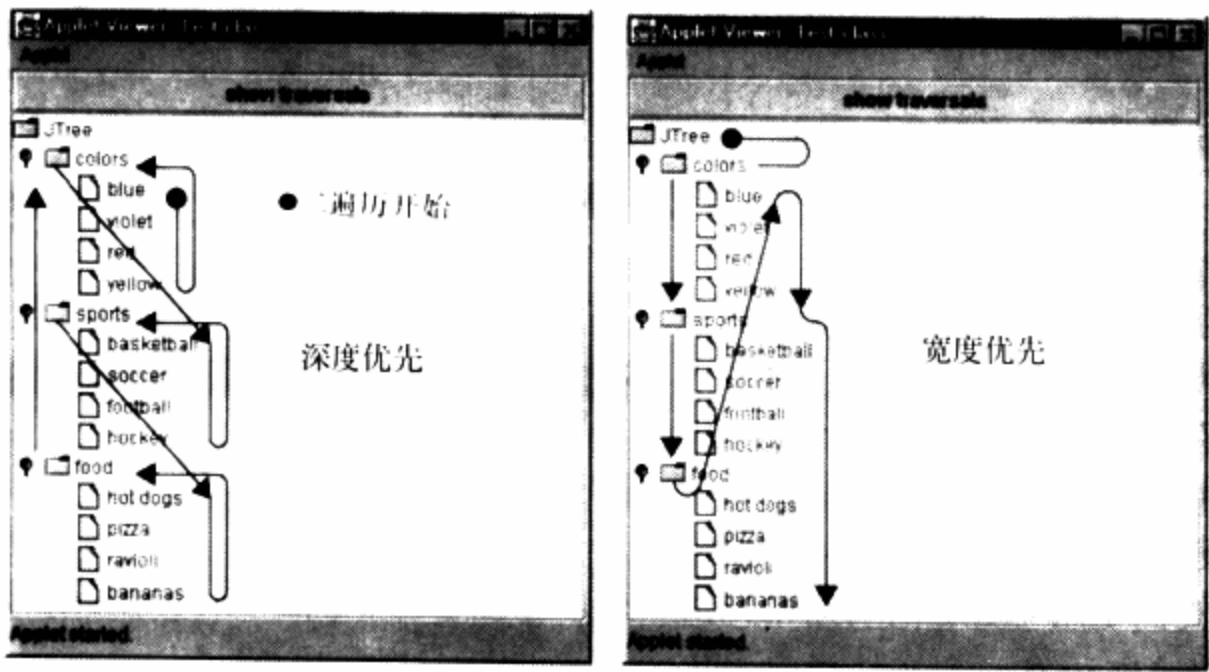


图 20-5 遍历树的顺序

MutableTreeNode.depthFirstEnumeration 和 DefaultMutableTreeNode.breadthFirstEnumeration 来获取的

例 20-3 深度优先遍历与宽度优先遍历之间的比较

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;
import javax.swing.tree.* ;
import java.util.* ;
```

```

public class Test extends JApplet {
    private JTree tree = new JTree ();
    private JButton button = new JButton ("show traversals");
    private DefaultMutableTreeNode root =
        (DefaultMutableTreeNode) tree.getModel ().getRoot ();

    public void init () {
        getContentPane ().add (new JScrollPane (tree),
                                BorderLayout.CENTER);
        getContentPane ().add (button, BorderLayout.NORTH);
        button.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                Enumeration df = root.depthFirstEnumeration ();
                Enumeration bf = root.breadthFirstEnumeration ();

                while (df.hasMoreElements ()) {
                    System.out.println (
                        df.nextElement ().toString ());
                }
                System.out.println ("");
                System.out.println ("");
                while (bf.hasMoreElements ()) {
                    System.out.println (
                        bf.nextElement ().toString ());
                }
            }
        });
    }
}

```

2. 扩展 DefaultMutableTreeNode

可以对目录和文件提供导航的文件查看器可能是树组件最自然的应用，图 20-6 中所示的应用程序就包含一个可用作文件查看器的 JTree 的一个实例。

Swing 树的设计基本是简便易行的，图 20-6 所示的应用程序就是一个证明，它有一个简单的实现方法。树包含有定制节点，这些节点是 FileNode 类的一些实例，而 FileNode 类维护作为自己的用户对象的一个 File 实例。

FileNode 类是 DefaultMutableTreeNode 的简单扩展，它的实例是用对 File 的引用来构造的。FileNode 构造方法将一个文件指定为该节点的用户对象。

FileNode 类重载 DefaultMutableTreeNode 类的 getAllows Children () 方法和 isLeaf () 方法。如果 FileNode 代表的是目录，则允许有子节点。如果它代表的不是目录，则它代表一个树叶。

FileNode 类还提供了一些方便的方法，以便获取对节点代表的文件的引用、确定节

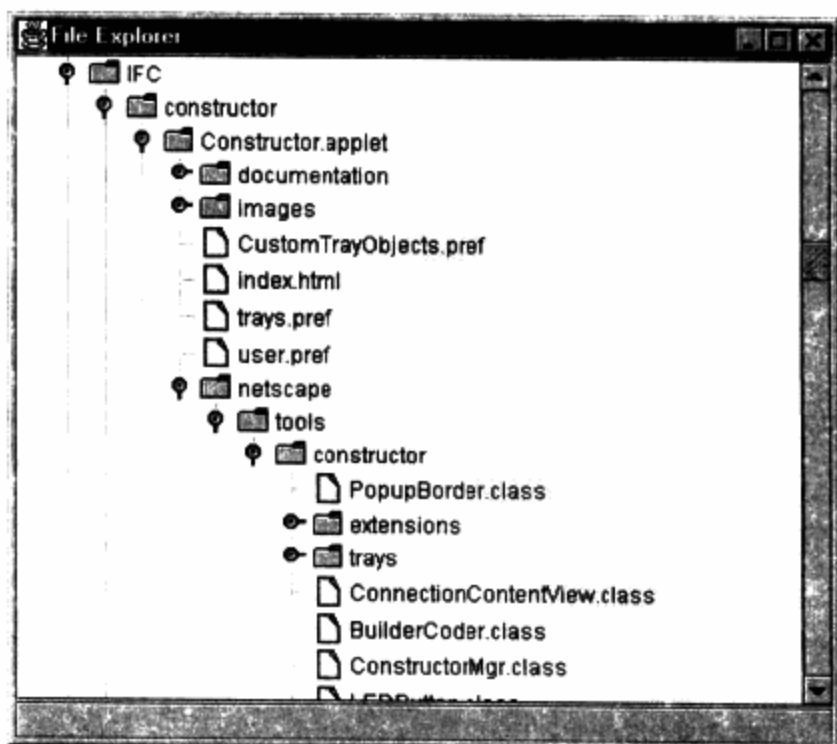


图 20-6 JTree 文件查看器

点是否代表目录和查看一个节点的子节点等等。需要说明的是，一个目录只能被查看一次。如果想多次查看一个目录，在实现上就需要多下些功夫。

FileNode 类还重载 toString() 方法，以便返回路径的最后组件，这个路径由该节点的文件来表示。有关树的路径的详细内容，请参见 20.3 节“树路径”。

```
class FileNode extends DefaultMutableTreeNode {
    private boolean explored = false;

    public FileNode (File file) {
        setUserObject (file);
    }

    public boolean getAllowsChildren () { return isDirectory (); }
    public boolean isLeaf () { return ! isDirectory (); }
    public File getFile () { return (File) getUserObject (); }

    public boolean isExplored () { return explored; }

    public boolean isDirectory () {
        File file = (File) getUserObject ();
        return file.isDirectory ();
    }

    public String toString () {
        File file = getFile ();
        String filename = file.toString ();
        int index = filename.lastIndexOf ("\\");

        return (index != -1 && index != filename.length () - 1) ?
            filename.substring (index + 1) :
            filename;
    }

    public void explore () {
        if (! isDirectory ())
            return;

        if (! isExplored ()) {
            File file = getFile ();
            File [] children = file.listFiles ();

            for (int i = 0; i < children.length; ++ i)
                add (new FileNode (children [i]));

            explored = true;
        }
    }
}
```

实现了 FileNode 类以后，其他需要做的就是创建一个根节点（代表要查看的驱动器）。然后，通过实例化 DefaultTreeModel（指定为这个树的模型）的一个实例，把该节点指定为这个树的根节点。把一个树扩展监听器添加到这个树中，以便在展开未查看过的节点时查看这些节点（有关 TreeExpansionListener 的详细内容，请参见 20.8.6 节）。

```
...
public class Test extends JFrame {
    public Test () {
        final JTree tree = new JTree (createTreeModel ());
        JScrollPane scrollPane = new JScrollPane (tree);

        getContentPane ().add (scrollPane, BorderLayout.CENTER);
        ...
    }
}
```

```

tree.addTreeExpansionListener (new TreeExpansionListener () {
    public void treeCollapsed (TreeExpansionEvent e) {
        // must implement because Swing does not provide
        // event adapters like the AWT
    }

    public void treeExpanded (TreeExpansionEvent e) {
        TreePath path = e.getPath ();
        FileNode node = (FileNode)
            path.getLastPathComponent ();

        if (! node.isExplored () ) {
            DefaultTreeModel model = (
                DefaultTreeModel) tree.getModel ();
            ...
            node.explore ();
            model.nodeStructureChanged (node);
            ...
        }
    }
});

private DefaultTreeModel createTreeModel () {
    File root = new File ("E: /");
    FileNode rootNode = new FileNode (root), node;

    rootNode.explore ();
    return new DefaultTreeModel (rootNode);
}

```

树扩展监听器通过获取与这个展开事件相关联的树路径来获取正在被展开的节点。然后，利用 `TreePath.getLastPathComponent` 方法从树路径中提取出 `FileNode` 实例。

节点被查看后，就会产生对树模型的 `nodeStructureChanged` 方法的一个调用，该方法激发一个事件，指出有子节点添加到了节点上或从节点上删除了。有关树事件的详细内容，请参见 20.8.6 节“树事件”。

3. 题外话

图 20-6 中所示的文件查看器有一个特性，我们在前面的各节中都没有讨论。当你正在查看一个目录时，应用程序的状态区就会用“正在查看……”这样的字符串予以更新，如图 20-7 所示。

一般情况下，查看持续的时间都很短，因此，如果查看持续的时间有多长，状态区的字符串就显示多长时间的话，那么就几乎看不见字符串。解决的办法是，由树的展开监听器创建一个独立线程，在延迟 450 毫秒以后再清除状态区。

```

...
tree.addTreeExpansionListener (new TreeExpansionListener () {

```

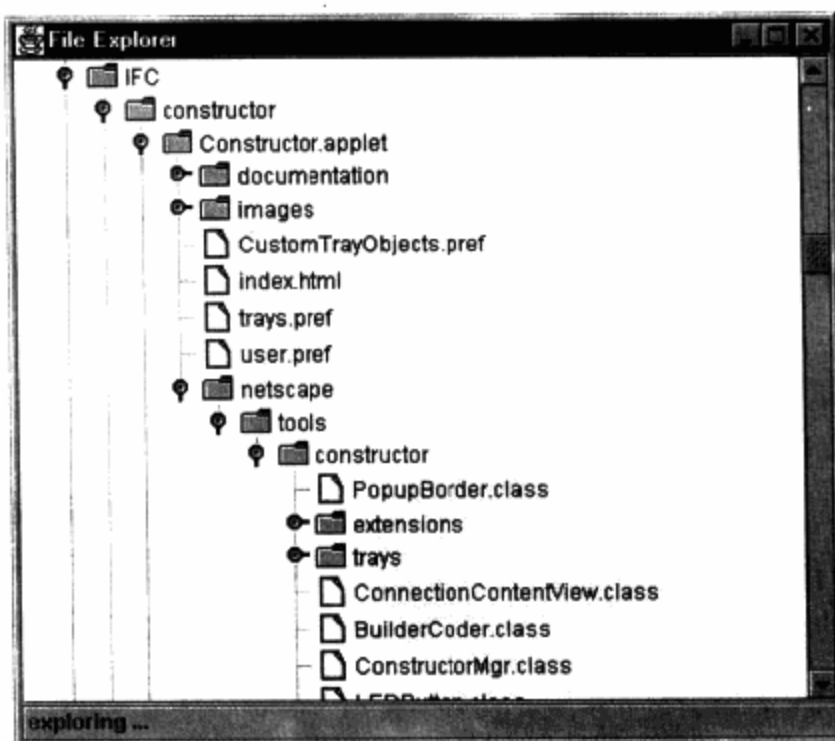


图 20-7 一个 JTree 的文件查看器

在延迟 450 毫秒以后再清除状态区。

```

public void treeCollapsed (TreeExpansionEvent e) {
    ...
}

public void treeExpanded (TreeExpansionEvent e) {
    UpdateStatus updateThread;
    TreePath path = e.getPath ();
    FileNode node = (FileNode)
        Path.getLastPathComponent ();

    If ( ! node.isExplored () ) {
        ...
        GJApp.updateStatus ("exploring...");
        UpdateStatus us = new UpdateStatus ();
        us.start ();
        node.explore ();
        ...
    }
}
...

```

在查看一个节点之前，用“正在查看……”字符串来更新这个应用程序的状态区，并且创建和启动一个 UpdateStatus 线程。

```

...
class UpdateStatus extends Thread {
    public void run () {
        try {Thread.currentThread ().sleep (450);}
        catch (InterruptedException e) {}

        SwingUtilities.invokeLater ( new Runnable () {
            Public void run () {
                GJApp.updateStatus ("");
            }
        });
    }
}
...

```

UpdateStatus 线程的 run 方法延迟 450 毫秒，然后调用 SwingUtilities.invokeLater () 方法来把一个 Runnable 放在事件派发线程上，该线程清除这个应用程序的状态区[○]。有关 Swing 多线程和 SwingUtilities.invokeLater 方法的介绍，请参见前面的 2.4 节“Swing 和线程”。

使用 UpdateStatus 的一个实例来清除应用程序状态区的结果是，每次查看一个节点时，“正在查看……”字符串大约显示半秒种。

例 20-4 列出了图 20-6 和图 20-7 中所示应用程序的完整代码。

例 20-4 JTree 文件查看器

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import java.awt.*;

```

○ UpdateStatus 线程不是事件派发线程，因此不能直接安全地更新 Swing 组件。


```

import java.awt.event.*;
import java.io.File;
import java.util.EventObject;

public class Test extends JFrame {
    public Test () {
        final JTree tree = new JTree (createTreeModel ());
        JScrollPane scrollPane = new JScrollPane (tree);

        getContentPane ().add (scrollPane, BorderLayout.CENTER);
        getContentPane ().add (GJApp.getStatusArea (),
                                BorderLayout.SOUTH);

        tree.addTreeExpansionListener (new TreeExpansionListener () {
            public void treeCollapsed (TreeExpansionEvent e) {
                |
            }
            public void treeExpanded (TreeExpansionEvent e) {
                UpdateStatus updateThread;
                TreePath path = e.getPath ();
                FileNode node = (FileNode)
                    path.getLastPathComponent ();

                if ( ! node.isExplored ()) {
                    DefaultTreeModel model =
                        (DefaultTreeModel) tree.getModel ();

                    GJApp.updateStatus ("exploring ...");

                    UpdateStatus us = new UpdateStatus ();
                    us.start ();

                    node.explore ();
                    model.nodeStructureChanged (node);
                }
            }
        });

        class UpdateStatus extends Thread {
            public void run () {
                try { Thread.currentThread ().sleep (450); }
                catch (InterruptedException e) {}

                SwingUtilities.invokeLater (new Runnable () {
                    public void run () {
                        GJApp.updateStatus ("");
                    }
                });
            }
        }

        private DefaultTreeModel createTreeModel () {
            File root = new File ("E: /");
            FileNode rootNode = new FileNode (root), node;

            rootNode.explore ();
            return new DefaultTreeModel (rootNode);
        }

        public static void main (String args []) {
            GJApp.launch (new Test (), "JTree File Explorer",
                          300, 300, 450, 400);
        }
    }
}

```

```

    }
}

class FileNode extends DefaultMutableTreeNode {
    private boolean explored = false;

    public FileNode (File file) {
        setUserObject (file);
    }

    public boolean getAllowsChildren () { return isDirectory (); }
    public boolean isLeaf () { return ! isDirectory (); }
    public File getFile () { return (File) getUserObject (); }

    public boolean isExplored () { return explored; }

    public boolean isDirectory () {
        File file = getFile ();
        return file.isDirectory ();
    }

    public String toString () {
        File file = (File) getUserObject ();
        String filename = file.toString ();
        int index = filename.lastIndexOf ("\\");

        return (index != -1 && index != filename.length () - 1) ?
            filename.substring (index + 1) :
            filename;
    }

    public void explore () {
        if (! isExplored ()) {
            File file = getFile ();
            File [] children = file.listFiles ();

            for (int i=0; i < children.length; ++i)
                add (new FileNode (children [i]));

            explored = true;
        }
    }
}

```

20.3 树路径

我们经常用树路径来确定树节点的数量。例如，当选取树的一个节点时，就用 `TreePath` 的一个实例来标识这个选取。`TreePath` 类标识一组节点，这些节点从一个节点到另外一个节点组成了一条路径。

图 20-8 中所示的应用程序有一个配备选取监听器的树，选取监听器在应用程序的状态区内显示最后选取的节点的路径。

一个选取监听器被添加到这个树中，这个选取监听器获取与最后选取的节点相关联的路径。然后，用这个树路径来更新该应用程序的状态区。



图 20-8 树路径

```

...
tree.addTreeSelectionListener (new TreeSelectionListener () {
    public void valueChanged (TreeSelectionEvent e) {
        TreePath path = e.getNewLeadSelectionPath ();
        if (path != null)
            GJApp.showStatus (" Path: " + path.toString ());
    }
});
...

```

除了上面讨论的树选取监听器以外，图 20-8 所示的应用程序和前面例 20-4 中列出的应用程序是一样的，因此这里没有给出图 20-8 所示应用程序的代码，其全部程序代码在书后所附的光盘上可以找到。

TreePath 类是 Object 的一个简单扩展，Object 维护代表一个路径的一组对象。类总结 20-1 总结了 TreePath 类。

类总结 20-1 TreePath

扩展：java.lang.Object

实现：java.io.Serializable

1. 构造方法：

```

protected TreePath ()
public TreePath (Object singlePath)
public TreePath (Object [] path Objects)
protected TreePath (Object [] path Objects, int length)
protected TreePath (TreePath parentPath, Object lastPathComponent)

```

大多数树路径由 Swing 类进行实例化，因此，开发人员很少需要构造 TreePath 的实例。TreePath 类提供了五个构造方法，这些构造方法以这样或那样的形式来指定路径。

无参数构造方法主要是为 TreePath 类的扩展而提供的，这些扩展希望存储路径对象的方式与在 TreePath 类中存储路径对象的方式不同。

如果向 Tree Path 传送 null 引用或空数组，它们的所有构造方法将弹出错误信息。

2. 方法

```

public Object getLastPathComponent ()
public TreePath getParentPath ()
public Object [] getPath ()
public Object getPathComponent (int index)
public int getPathCount ()
public TreePath pathByAddingChild (Object child)
public boolean equals (Object)
public boolean isDescendant (TreePath)
public int hashCode ()
public String toString ()

```

上面列出的第一组方法都返回有关树路径的信息。

最常用的方法无疑是 getLastPathComponent，它返回对路径中最后一个对象的引用。图 20-8 中所示的应用程序中就使用了 getLastPathComponent 方法。

上面列出的第二组方法是一些方便的方法，它们判断一个树路径是否等同于一个对象（假定是另外一条树路径），或某条给定的树路径是否是另外一条路径的后代。如果一条路径的每

个对象等于另一个路径中每个相应的对象，则这两条路径是相同的。

上面列出的最后两个方法是对 `Object` 的重载。`TreePath` 的哈希代码等于这条路径中最后对象的哈希代码。

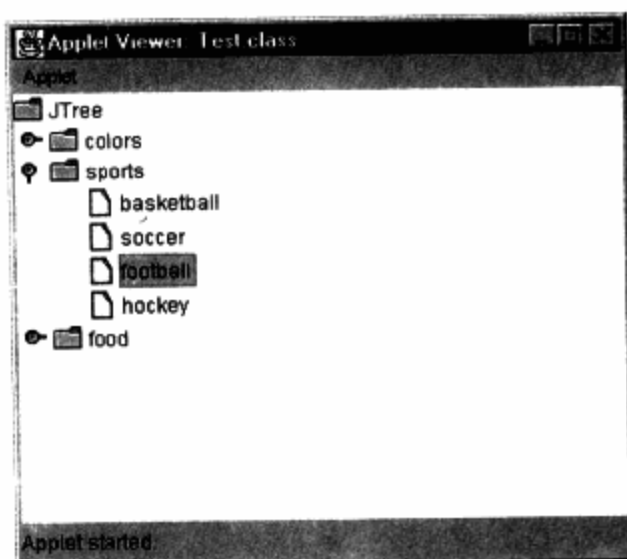


图 20-9 中所示的小应用程序包含了一个配备选取监听器的树，这个选取监听器显示由许多 `TreePath` 方法获取的信息。

例 20-5 列出了图 20-9 所示小应用程序的代码。

图 20-9 使用树路径

例 20-5 使用树路径

```
import javax.swing.*.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import java.awt.*.*;

public class Test extends JApplet {
    JTree tree = new JTree ();
    DefaultTreeModel model = (DefaultTreeModel) tree.getModel ();
    TreeSelectionModel selectionModel = tree.getSelectionModel ();

    public void init () {
        getContentPane ().add (tree, BorderLayout.CENTER);
        tree.addTreeSelectionListener (
            new TreeSelectionListener () {
                public void valueChanged (TreeSelectionEvent e) {
                    TreePath path = e.getNewLeadSelectionPath ();
                    if (path == null)
                        System.out.println ("Selection Cleared");
                    else {
                        TreePath parentPath = path.getParentPath ();
                        Object
                            .lastNode = path.getLastPathComponent (),
                            firstNode = path.getPathComponent (0);

                        System.out.println ("Path: " + path +
                            " has " +
                                path.getPathCount () +
                                " nodes");
                        System.out.println ("Last Path Component: " +
                            lastNode.toString ());
                        System.out.println ("First Path Component: " +
                            firstNode.toString ());
                        System.out.println ("Parent Path: " +
                            parentPath);

                        // the following if statement is always true
                        if (parentPath.isDescendant (path) ) {
                            System.out.println (parentPath +
                                " is a descendant of " + path);
                        }
                    }
                }
            }
        );
    }
}
```


TreeModel 接口为树的根节点定义了访问方法。上面列出的第二组方法是子节点、节点数和某个特定子节点的索引的方法。应该注意的是，从节点本身也可以直接获取同样的信息。实际上，DefaultTreeModel 把这个任务交给传送这些方法的父节点来处理。

3. 树叶标识/改变节点值

```
public abstract boolean isLeaf (Object node)
```

```
public abstract void valueForPathChanged (TreePath, Object node)
```

TreeModel 接口定义了一个 isLeaf 方法，用以确定一个特定的节点是否是树叶。树节点还提供了 isLeaf 方法，然而缺省的树模型可能会根据节点是否允许有子节点来确定节点是否是树叶。

valueForPathChanged 方法为路径中的最后一个节点设置用户对象，并激发一个树模型事件，以指出这个节点发生了变化。

DefaultTreeModel

DefaultTreeModel 类是 Swing 对 TreeModel 接口的单独实现。DefaultTreeModel 类除了提供许多其他的方法外，还实现了由 TreeModel 接口定义的方法。

图 20-10 示出了 DefaultTreeModel 类的类图。

DefaultTreeModel 类扩展 Object 类，并实现了 TreeModel 接口和 Serializable 接口。EventListenerList 的实例维护树模型监听器的列表，而这个监听器列表维护代表这个树的根节点的一个 TreeNode 引用。

我们来做一下回顾。TreeModel 接口定义了 isLeaf 方法，该方法确定一个节点是树叶还是文件夹。DefaultTreeModel 实现了 isLeaf 方法，如下所示：

```
// from DefaultTreeModel.java ...
```

```
public boolean isLeaf (Object node) {
    if (asksAllowsChildren)
        return ! ((TreeNode) node).getAllowsChildren ();
    return ((TreeNode) node).isLeaf ();
}
```

如果这个模型的 asksAllowsChildren 属性为 true，那么就根据节点是否允许有子节点来确定节点的树叶状态。否则，如果节点的 isLeaf 方法返回值为 true，那么它就是树叶。如果 asksAllowsChildren 属性为 false，而节点的 isLeaf 方法返回值为 true，那么这个节点就是一个文件夹。

类总结 20-2 中对 DefaultTreeModel 类进行了总结。

类总结 20-2 DefaultTreeModel

扩展：Object

实现：TreeModel, Serializable

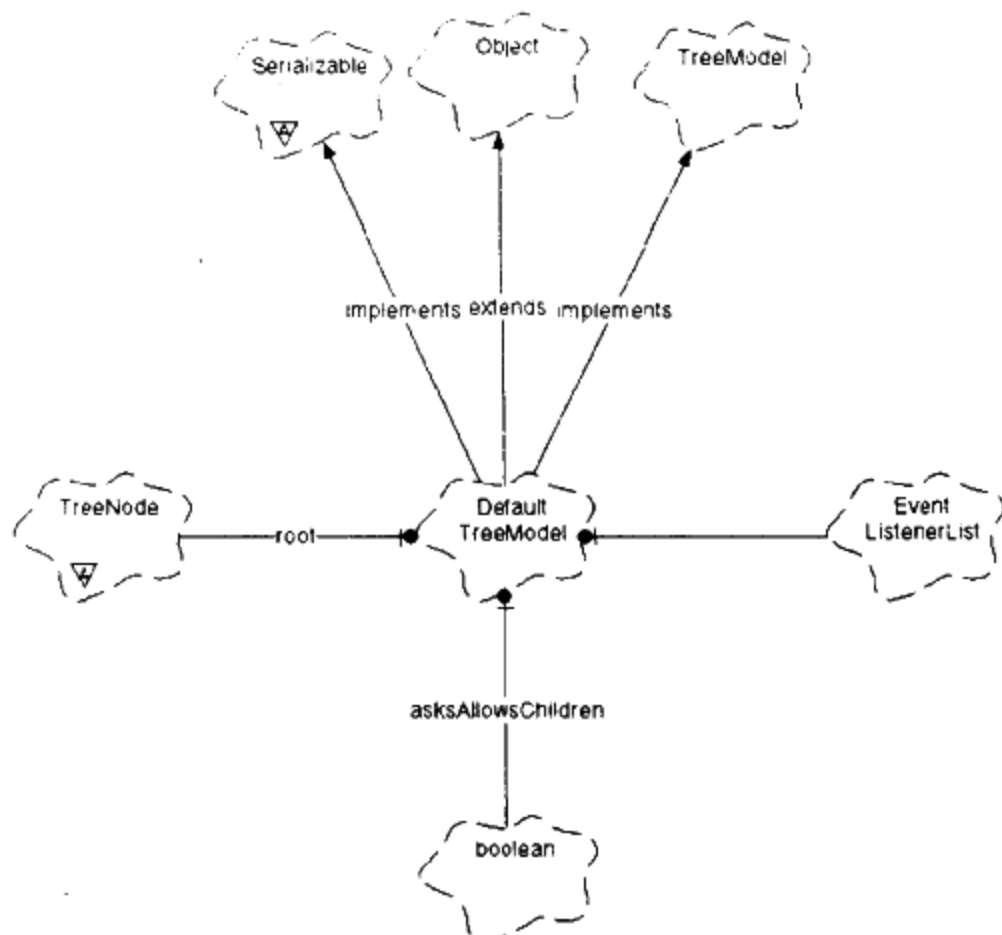


图 20-10 DefaultTreeModel 类的类图

1. 构造方法:

```
public DefaultTreeModel (TreeNode root)
```

```
public DefaultTreeModel (TreeNode root, boolean asksAllowsChildren)
```

DefaultTreeModel 类提供了两个构造方法, 都需要向它们传送树的根节点。asksAllowsChildren boolean 变量控制模型是否通过询问节点允许或不允许有子节点来确定节点的树叶状态。

注意, DefaultTreeModel 没有提供无参数构造方法, 这就意味着在构造时必须指定一个根节点。

2. 方法

(1) TreeModel 方法

```
public void addTreeModelListener (TreeModelListener)
```

```
public void removeTreeModelListener (TreeModelListener)
```

```
public Object getChild (Object parent, int index)
```

```
public int getChildCount (Object parent)
```

```
public int getIndexOfChild (Object parent, Object child)
```

```
public void getRoot (TreeNode)
```

```
public void valueForPathChanged (TreePath, Object)
```

上面列出的这些方法均由 TreeModel 接口定义。第二组方法把任务直接交给传送给它们的父节点来完成。

(2) 激发事件

```
protected void fireTreeNodesChanged (Object, Object [], int [], Object [])
```

```
protected void fireTreeNodesInserted (Object, Object [], int [], Object [])
```

```
protected void fireTreeNodesRemoved (Object, Object [], int [], Object [])
```

```
protected void fireTreeStructureChanged (Object, Object [], int [], Object [])
```

与许多 Swing 模型一样, DefaultTreeModel 类也提供了用以激发事件的一组方法。当改变、插入或删除节点, 或者修改了树的结构时, DefaultTreeModel 的扩展都必须激发相应的事件。

DefaultTreeModel 的扩展可以使用上面列出的方法来激发事件, 以便登记树模型监听器。这些方法还被 DefaultTreeModel 在内部使用。

(3) 设置根/到根的路径/插入和删除节点

```
public Object setRoot ( )
```

```
public TreeNode [ ] getPathToRoot (TreeNode)
```

```
protected TreeNode [ ] getPathToRoot (TreeNode, int depth)
```

```
public void insertNodeInto (MutableTreeNode, MutableTreeNode, int index)
```

```
public void removeNodeFromParent (MutableTreeNode)
```

DefaultTreeModel 除了提供对代表一条路径的 TreeNodes 数组的访问方法外, 还提供了对根节点的访问方法。带有一个 integer 值的 getPathToRoot 方法被递归调用, 但不能在 DefaultTreeModel 类 (或其扩展) 之外调用这个 getPathToRoot 方法。

可用上面列出的最后两个方法向父节点插入子节点, 或将子节点从它的父节点中删去。这两个方法激发树模型事件, 因此比类似的 DefaultMutableTreeNode 方法更让人乐于使用。

(4) 通知方法

```
public void nodeChanged (TreeNode)
```

```
public void nodeStructureChanged (TreeNode)
```

```
public void nodesChanged (TreeNode, int [])
```

```
public void nodesWereInserted (TreeNode, int [])
```

```
public void nodesWereRemoved (TreeNode, int [], Object [])
```

上面列出的方法用来创建相应的事件, 并把这些事件发送给已登记的树模型监听器。DefaultTreeModel 内部地使用这些方法来激发事件。例如, DefaultTreeModel.insertNodeInto () 在插

入节点后就会调用 `nodeWereInserted()`。
(5) 重载/是否允许有子节点/是否是树叶

```
public void reload ( )
public void reload (TreeNode)
public void setAsksAllowsChildren (boolean)
public boolean asksAllowsChildren ( )
public boolean isLeaf (Object)
```

`reload` 方法激发事件，指出某个给定节点的子节点已被修改。无参数重载方法以根节点为参数来调用 `reload (TreeNode)`，指出树中的所有节点可能都已被修改。

`reload()` 方法的调用将使树中所有展开的文件夹折叠起来，整个树被重画。

上面列出的第二组方法可访问 `DefaultTreeModel` 的 `asksAllowsChildren` 属性。如果调用了 `setAsksAllowsChildren (true)`，`DefaultTreeModel.isLeaf()` 就将根据节点是否允许有子节点来确定它是否是树叶。如果 `asksAllowsChildren` 属性是 `false` (这是该属性的缺省值)，`DefaultTreeModel.isLeaf()` 就返回节点的 `isLeaf` 方法的返回值。

图 20-11 中所示的应用程序说明的是，向 (从) 一个树模型中添加和删除节点。图中有两个按钮，用于添加和删除节点。

从图 20-11 左上角的图片可以看到，`Add node` (添加节点) 按钮的动作监听器显示一个对话框，提示输入节点名。新创建的节点添加在当前选取的节点之下。图 20-11 右上角和左下角的图片说明，树模型的模型监听器显示一个对话框，以获知哪个节点被添加或删除了。

按钮的允许状态是由添加到这个树中的树选取监听器设置的。只有选取了树中的一个节点后，这两个按钮才是允许的。如果没有节点被选中，则两个按钮都是不允许的，图 20-11 中右下角的图片说明了这一点。

图 20-11 中的应用程序创建 `JTree` 的一个实例，并获取对这个树的模型的一个引用。需要说明的是，树模型监听器必须实现由 `TreeModelListener` 接口定义的所有方法，这是因为 `Swing` 的

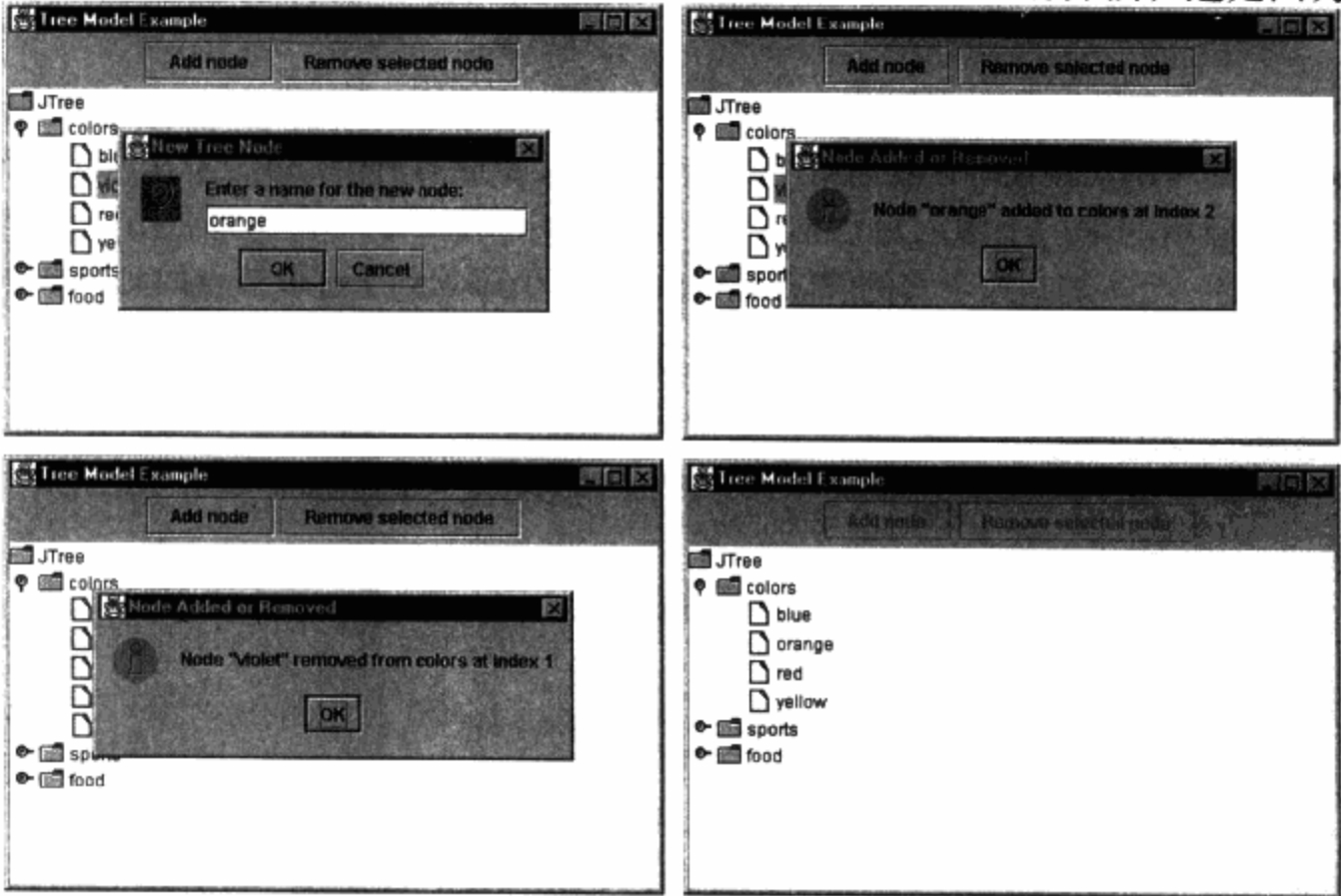


图 20-11 插入和删除树节点

1.1FCS 版本并不像 AWT 一样提供事件适配器类。

```
public class Test extends JFrame {
    JTree tree = new JTree ();
    DefaultTreeModel model = (DefaultTreeModel) tree.getModel ();

    ...

    public Test () {
        ...
        model.addTreeModelListener (new TreeModelListener () {
            public void treeNodesInserted (TreeModelEvent e) {
                showInsertionOrRemoval (e, " added to ");
            }
            public void treeNodesRemoved (TreeModelEvent e) {
                showInsertionOrRemoval (e, " removed from ");
            }
            private void showInsertionOrRemoval (TreeModelEvent e,
                                                String s) {
                Object [] parentPath = e.getPath ();
                int [] indexes = e.getChildIndices ();
                Object [] children = e.getChildren ();
                Object parent = parentPath [parentPath.length-1];
                JOptionPane.showMessageDialog (Test.this,
                    "Node \ "" + children [0] .toString () +
                    " \ "" + s + parent.toString () +
                    " at index " + indexes [0],
                    "Node Added or Removed",
                    JOptionPane.INFORMATION_MESSAGE);
            }
            public void treeNodesChanged (TreeModelEvent e) {}
            public void treeStructureChanged (TreeModelEvent e) {}
        });
    }
    ...
}
```

这个应用程序的按钮被包含在 Control Panel 的一个实例中，这个实例是 JPanel 类的扩展。添加到 Add node 按钮中的动作监听器获取对所选取节点的父节点的引用，并计算出所选取节点之后的索引值。随后显示的是一个输入对话框，提示输入节点名。输入的节点名用于创建 DefaultMutableTreeNode 的一个实例，这个实例就以计算出的索引值添加到所选取节点的父节点中。

```
class ControlPanel extends JPanel {
    public ControlPanel () {
        ...
        addButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                TreePath path =
                    selectionModel.getSelectionPath ();

                MutableTreeNode parent, node =
                    (MutableTreeNode) path.getLastPathComponent ();

                if (path.getPathCount () > 1)
                    parent = (MutableTreeNode) node.getParent ();
                else
                    parent = (MutableTreeNode) node; // root node
            }
        });
    }
}
```

```

        int index = parent.getIndex (node) + 1;
        String s = JOptionPane.showInputDialog (
            Test.this,
            "Enter a name for the new node:",
            "New Tree Node",
            JOptionPane.QUESTION_MESSAGE);
        MutableTreeNode newNode =
            new DefaultMutableTreeNode (s);
        model.insertNodeInto (newNode, parent, index);
    }
}
...

```

Remove selected node 按钮也添加了一个动作监听器。这个监听器获取对选取节点的一个引用，并调用模型的 removeNodeFromParent () 方法。

```

...
removeButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        TreePath path =
            selectionModel.getSelectionPath ();

        if (path.getPathCount () == 1) { // root node
            JOptionPane.showMessageDialog (
                ControlPanel.this,
                "Can't remove root node!");
            return;
        }

        MutableTreeNode node =
            (MutableTreeNode) path.getLastPathComponent ();
        model.removeNodeFromParent (node);
    }
});
}
}

```

例 20-6 列出了图 20-11 所示应用程序的完整代码。

例 20-6 添加和删除节点

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    JTree tree = new JTree ();
    DefaultTreeModel model = (DefaultTreeModel) tree.getModel ();
    TreeSelectionModel selectionModel = tree.getSelectionModel ();
    JButton removeButton = new JButton ("Remove selected node");
    JButton addButton = new JButton ("Add node");

    public Test () {
        Container contentPane = getContentPane ();

        selectionModel.setSelectionMode (

```

```

        TreeSelectionMode.SINGLE_TREE_SELECTION);

contentPane.add (new ControlPanel (), BorderLayout.NORTH);
contentPane.add (tree, BorderLayout.CENTER);

tree.addTreeSelectionListener (
    new TreeSelectionListener () {
        public void valueChanged (TreeSelectionEvent e) {
            TreePath path = e.getNewLeadSelectionPath ();
            boolean nodesAreSelected = (path != null);
            addButton.setEnabled (nodesAreSelected);
            removeButton.setEnabled (nodesAreSelected);
        }
    });

model.addTreeModelListener (new TreeModelListener () {
    public void treeNodesInserted (TreeModelEvent e) {
        showInsertionOrRemoval (e, " added to ");
    }
    public void treeNodesRemoved (TreeModelEvent e) {
        showInsertionOrRemoval (e, " removed from ");
    }
    private void showInsertionOrRemoval (TreeModelEvent e,
        String s) {
        Object [] parentPath = e.getPath ();
        int [] indexes = e.getChildIndices ();
        Object [] children = e.getChildren ();
        Object parent = parentPath [parentPath.length-1];
        JOptionPane.showMessageDialog (Test.this,
            "Node \ \"" + children [0] .toString () +
            " \ \"" + s + parent.toString () +
            " at index " + indexes [0],
            "Node Added or Removed",
            JOptionPane.INFORMATION_MESSAGE);
    }
    public void treeNodesChanged (TreeModelEvent e) {}
    public void treeStructureChanged (TreeModelEvent e) {}
});

class ControlPanel extends JPanel {
    public ControlPanel () {
        addButton.setEnabled (false);
        removeButton.setEnabled (false);

        add (addButton);
        add (removeButton);

        addButton.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                TreePath path =
                    selectionModel.getSelectionPath ();
                MutableTreeNode parent, node =
                    (MutableTreeNode) path.getLastPathComponent ();
                if (path.getPathCount () > 1)
                    parent = (MutableTreeNode) node.getParent ();
                else

```

```

        parent = (MutableTreeNode) node;
        int index = parent.getIndex (node) + 1;
        String s = JOptionPane.showInputDialog (
            Test.this,
            "Enter a name for the new node:",
            "New Tree Node",
            JOptionPane.QUESTION_MESSAGE);

        MutableTreeNode newNode =
            new DefaultMutableTreeNode (s);
        model.insertNodeInto (newNode, parent, index);
    }
    });
    removeButton.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            TreePath path =
                selectionModel.getSelectionPath ();

            if (path.getPathCount () == 1) {
                JOptionPane.showMessageDialog (
                    ControlPanel.this,
                    "Can't remove root node!");
                return;
            }

            MutableTreeNode node =
                (MutableTreeNode) path.getLastPathComponent ();
            model.removeNodeFromParent (node);
        }
    });
}

public static void main (String args []) {
    GraphicJavaApplication.launch (new Test (),
        "Tree Model Example", 300, 300, 450, 300);
}

class GraphicJavaApplication extends WindowAdapter {
    public static void launch (final JFrame f, String title,
        final int x, final int y,
        final int w, int h) {
        f.setTitle (title);
        f.setBounds (x, y, w, h);
        f.setVisible (true);

        f.setDefaultCloseOperation (
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener (new WindowAdapter () {
            public void windowClosed (WindowEvent e) {
                System.exit (0);
            }
        });
    }
}

```

Swing 提示

是树叶，还是文件夹？

当确定一个节点的树叶状态时，需要用到两个属性：节点的 `allowsChildren` 属性和树模型的 `asksAllowsChildren` 属性。

如果一个树模型的 `asksAllowsChildren` 属性是 `true`，模型就根据节点的 `allowsChildren` 属性确定节点是不是树叶。如果节点允许有子节点，那么它就是文件夹。反之，不允许有子节点的节点就是树叶。如果树模型的 `asksAllowsChildren` 属性是 `false`，模型就利用节点的 `isLeaf` 属性来确定它是不是树叶。需要注意的是，确定节点树叶状态的机制提供了两级自由。换句话说，单个节点是树叶还是文件夹，可以根据与它相关联的树模型而变化。

20.5 树选取

树选取由 `TreeSelectionModel` 接口定义，接口总结 20-4 对 `TreeSelectionModel` 进行了总结。

接口总结 20-4 `TreeSelectionModel`

1. 常量

```
public static final int CONTIGUOUS TREE SELECTION
public static final int DISCONTIGUOUS TREE SELECTION
public static final int SINGLE TREE SELECTION
```

上面列出的这些常量代表树选取模式，连续选取允许在任何时候选取一组连续的节点。不连续选取允许选取一组以上的连续节点。单个选取将选取范围限定为单个节点。

缺省的树选取模式是 `DISCONTIGUOUS _ TREE _ SELECTION`。

2. 方法

(1) 选取模式

```
public abstract int getSelectionMode ()
public abstract void setSelectionMode (int)
```

上面列出的这些方法用以访问树选取模式，`setSelectionMode` 方法必须以上面所列的一个常量为参数。

(2) 监听器

```
public abstract void addPropertyChangeListener (PropertyChangeListener)
public abstract void addTreeSelectionListener (TreeSelectionListener)
public abstract void removePropertyChangeListener (PropertyChangeListener)
public abstract void removeTreeSelectionListener (TreeSelectionListener)
```

树选取模型激发属性改变事件和树选取事件。上面列出的这些方法允许监听器向这个选取模型登记。

在 20.8.6 节中将对树选取事件进行讨论。

(3) 选取路径

```
public abstract void addSelectionPath (TreePath)
public abstract void addSelectionPaths (TreePath [])

public abstract void setSelectionPath (TreePath)
public abstract void setSelectionPaths (TreePath [])

public abstract void removeSelectionPath (TreePath)
```

```
public abstract void removeSelectionPaths (TreePath [])
```

```
public abstract TreePath getSelectionMode ()
```

```
public abstract TreePath [] getSelectionPaths ()
```

```
public abstract TreePath getLeadSelectionPath ()
```

```
public abstract boolean isPathSelected (TreePath)
```

树选取是以选取路径的数组的形式指定的。上面所列的前三组方法用来添加、设置和删除选取路径。这些方法被 Swing 在内部使用以设置树选取，也可以由开发人员使用，以程序的形式来设置选取。

最后两组方法获取当前选取的有关信息。getSelectionPaths 方法返回当前选取的所有路径的一个数组，而 getSelectionPath () 返回的是第一个选取路径。getLeadSelectionPath 方法返回添加到选取中的最后路径，isPathSelected () 可以确定一条路径是否被选取。

(4) 选取行

```
public abstract int [] getSelectionRows ()
```

```
public abstract int getLeadSelectionRow ()
```

```
public abstract int getMaxSelectionRow ()
```

```
public abstract int getMinSelectionRow ()
```

```
public abstract boolean isRowSelected (int)
```

```
public abstract void setRowMapper (RowMapper)
```

```
public abstract RowMapper getRowMapper ()
```

上面列出的第一组方法返回选取行的有关信息。getSelectionRows 方法返回一个代表当前所有选取行的整型数组，而 getLeadSelectionRow () 返回添加到选取中的最后一行，getMaxSelectionRow 和 getMinSelectionRow 两个方法分别返回选取行的最大索引和最小索引。通过 isRowSelected 方法可以确定是否选取了一行。

每个树选取模型都有一个行映射器，将路径翻译为代表行的一个整型数组。TreeSelectionModel 接口提供了对行映射器的访问方法。在实际应用中，开发人员很少，甚至从不使用或扩展这些方法。

(5) 实用方法

```
public abstract int getSelectionCount ()
```

```
public abstract void clearSelection ()
```

```
public abstract boolean isSelectionEmpty ()
```

```
public abstract void resetRowSelection ()
```

TreeSelectionModel 接口定义了丰富的实用方法，使开发人员可以获取选取节点的数目、清除选取以及确定是否有节点被选取。

因为当展开或折叠文件夹的时候，选取行可能会发生变化，所以 resetRowSelection 方法就借助选取模型的行映射器来计算哪些行被选取。resetRowSelection 方法由 Swing 在内部调用。

DefaultTreeSelectionModel

Swing 以 DefaultTreeSelectionModel 类的形式提供了 TreeSelectionModel 接口的单个实现。图 20-12 是 DefaultTreeSelectionModel 类的类图。

DefaultTreeSelectionModel 实现 TreeSelectionModel 接口，并维护对列表选取模型、行映射器和事件监听器列表的引用。DefaultTreeSelectionModel 还维护一个 TreePath 数组和一个 TreePath，其中 TreePath 数组代表当前的选取，TreePath 代表前导路径（即添加到这个选取中的最后的路径）。为了选取模式、前导索引和行，还维护了一些整型值。

图 20-13 中所示的小应用程序说明了树选取模式和一个树选取模型的使用。这个小应用程序

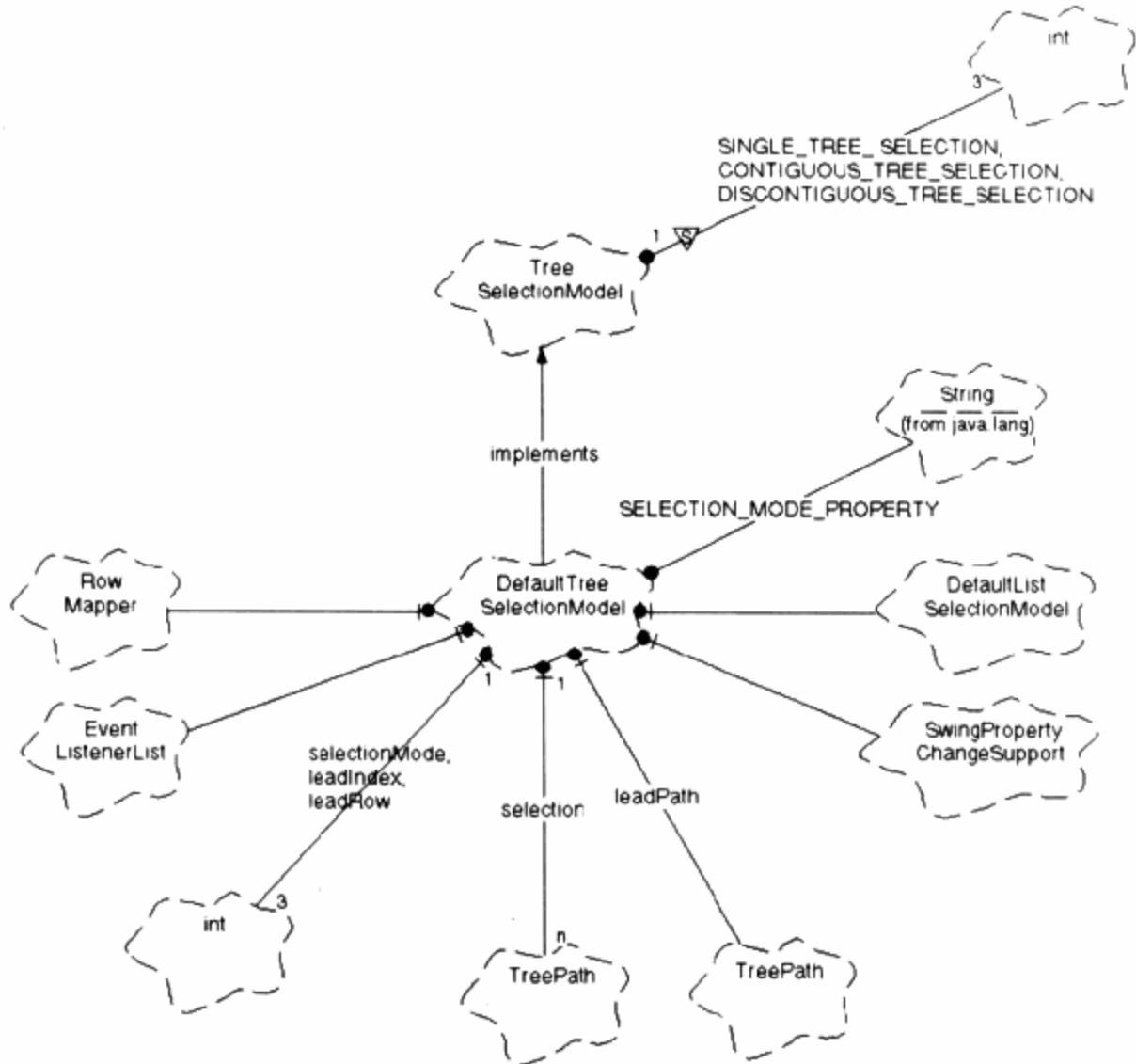


图 20-12 DefaultTreeSelectionModel 类的类图

序提供了一个组合框用以选取树选取模式，还提供了一个按钮用以清除这个选取。

图 20-13 所示的小应用程序用 JTree 无参数构造方法创建了一个树，并获取了对树选取模

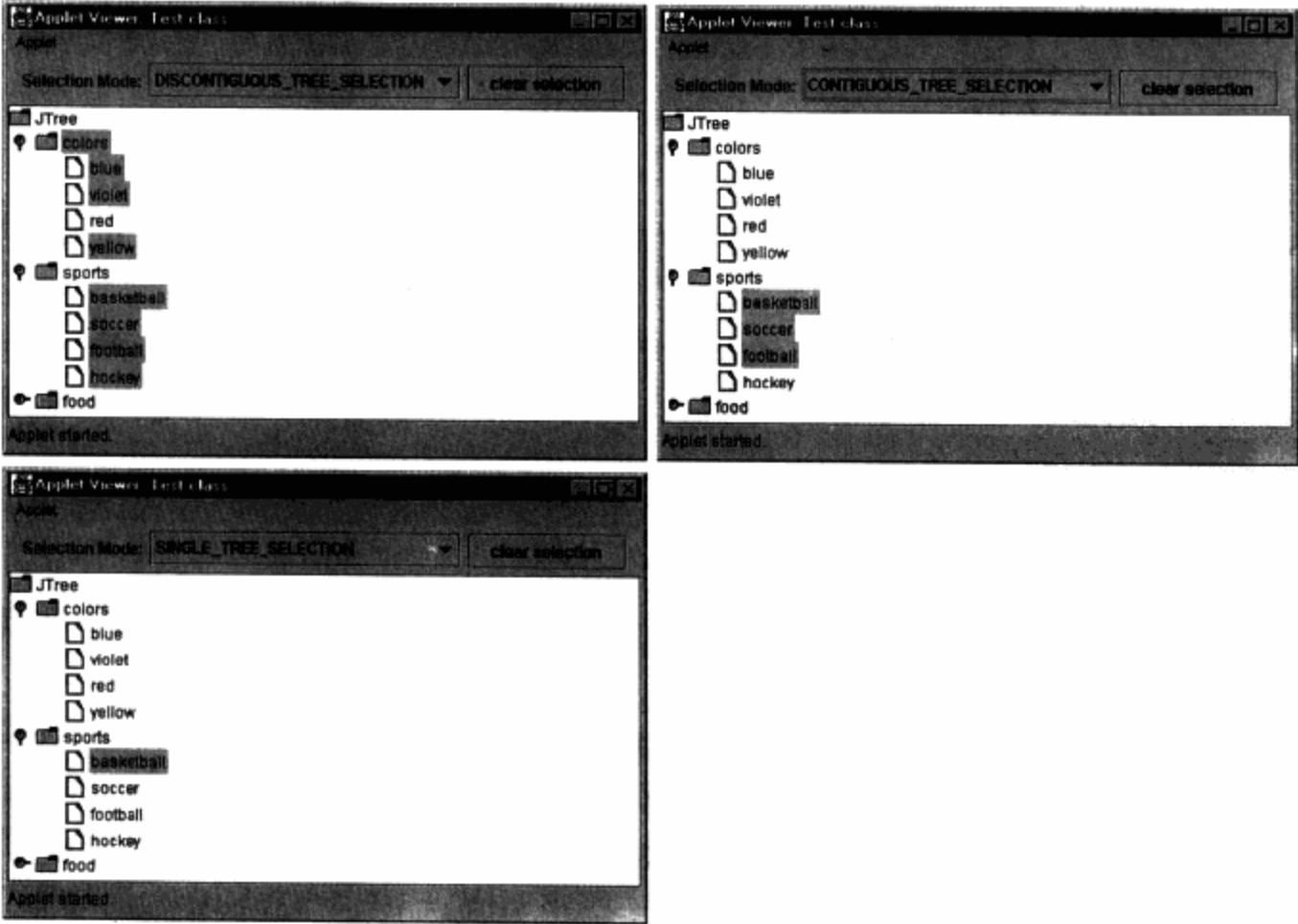


图 20-13 树选取模式

型的一个引用。它还定义了代表选取模式的字符串数组和整数数组。

```
public class Test extends JApplet {
    JTree tree = new JTree ( );
    TreeSelectionModel selectionModel = tree.getSelectionModel ( );
    String modes [ ] = {
        "CONTIGUOUS_TREE_SELECTION",
        "DISCONTIGUOUS_TREE_SELECTION",
        "SINGLE_TREE_SELECTION"
    };
    int modelIds [ ] = {
        TreeSelectionModel.CONTIGUOUS_TREE_SELECTION,
        TreeSelectionModel.DISCONTIGUOUS_TREE_SELECTION,
        TreeSelectionModel.SINGLE_TREE_SELECTION,
    };
    ...
}
```

图中所示小应用程序中的组合框包含在 `ControlPanel` 的一个实例中, `ControlPanel` 是 `JPanel` 的扩展。`ControlPanel` 构造方法根据初始选取模式初始化组合框,并向组合框添加动作监听器,以便更新树的选取模式。

```
...
class ControlPanel extends JPanel {
    JComboBox combo = new JComboBox ( );

    public ControlPanel ( ) {
        for (int i = 0; i < modes.length; ++i) {
            combo.addItem (modes [i]);
        }
        add (new JLabel ("Selection Mode:"));
        add (combo);

        int initialMode = selectionModel.getSelectionMode ( );
        if (initialMode == modelIds [0])
            combo.setSelectedIndex (0);
        else if (initialMode == modelIds [1])
            combo.setSelectedIndex (1);
        else if (initialMode == modelIds [2])
            combo.setSelectedIndex (2);

        combo.addActionListener (new ActionListener ( ) {
            public void actionPerformed (ActionEvent e) {
                int index = combo.getSelectedIndex ( );
                selectionModel.setSelectionMode (
                    modelIds [index]);
            }
        });
    }
    ...
}
```

在 `clear selection` 按钮中添加了一个动作监听器,以便调用树选取模型的 `clearSelection` 方法。

```
...
button.addActionListener (new ActionListener ( ) {
    public void actionPerformed (ActionEvent e) {
        selectionModel.clearSelection ( );
    }
});
```

例 20-7 列出了图 20-13 所示小应用程序的完整代码。

例 20-7 树选取模式

```
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    JTree tree = new JTree ();

    TreeSelectionModel selectionModel = tree.getSelectionModel ();

    String modes [] = {
        "CONTIGUOUS_TREE_SELECTION",
        "DISCONTIGUOUS_TREE_SELECTION",
        "SINGLE_TREE_SELECTION"
    };

    int modelIds [] = {
        TreeSelectionModel.CONTIGUOUS_TREE_SELECTION,
        TreeSelectionModel.DISCONTIGUOUS_TREE_SELECTION,
        TreeSelectionModel.SINGLE_TREE_SELECTION,
    };

    public void init () {
        Container contentPane = getContentPane ();

        contentPane.add (new ControlPanel (), BorderLayout.NORTH);
        contentPane.add (new JScrollPane (tree),
            BorderLayout.CENTER);
    }

    class ControlPanel extends JPanel {
        JComboBox combo = new JComboBox ();
        JButton button = new JButton ("clear selection");

        public ControlPanel () {
            for (int i=0; i < modes.length; ++i) {
                combo.addItem (modes [i]);
            }
            add (new JLabel ("Selection Mode:"));
            add (combo);
            add (button);

            int initialMode = selectionModel.getSelectionMode ();
            if (initialMode == modelIds [0])
                combo.setSelectedIndex (0);
            else if (initialMode == modelIds [1])
                combo.setSelectedIndex (1);
            else if (initialMode == modelIds [2])
                combo.setSelectedIndex (2);

            combo.addActionListener (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    int index = combo.getSelectedIndex ();
                    selectionModel.setSelectionMode (
                        modelIds [index]);
                }
            });
        }
    }
}
```

```
button.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        selectionModel.clearSelection ();
    }
});
```

20.6 树单元绘制

与其他 Swing 绘制器一样，树单元绘制器也是由一个接口定义的，这个接口只定义返回一个组件的方法。接口总结 20-5 对 TreeCellRenderer 接口进行了总结。

接口总结 20-5 TreeCellRenderer

```
public abstract Component getTreeCellRendererComponent (JTree tree,
                                                         Object value,
                                                         boolean selected,
                                                         boolean expanded,
                                                         boolean leaf,
                                                         int row,
                                                         boolean hasFocus)
```

使用由 getTreeCellRendererComponent 返回的组件就像使用橡皮图章一样，它把组件绘制到树的节点中。传送给这个方法的参数有：树、要绘制的值以及单元是否被选取、被展开、是不是树叶，有没有焦点等状态。需要传送的参数还有节点的行。

20.6.1 DefaultTreeCellRenderer

Swing . tree 包以 DefaultTreeCellRenderer 类的形式提供了一个缺省的绘制器。图 20-14 示出了 DefaultTreeCellRenderer 类的类图。

DefaultTreeCellRenderer 类扩展 JLabel，并实现 TreeCellRenderer 接口。这个类维护三个 Icon 引用，这三个引用用于树叶节点、打开的文件夹节点和关闭的文件夹节点。同时还维护文本颜色、背景和绘制器边框。

类总结 20-3 对 DefaultTreeCellRenderer 类进行了总结。

类总结 20-3 DefaultTreeCellRenderer

1. 构造方法

```
public DefaultTreeCellRenderer ()
```

DefaultTreeCellRenderer 类只提供了一个构造方法，这个无参数构造方法将绘制器的

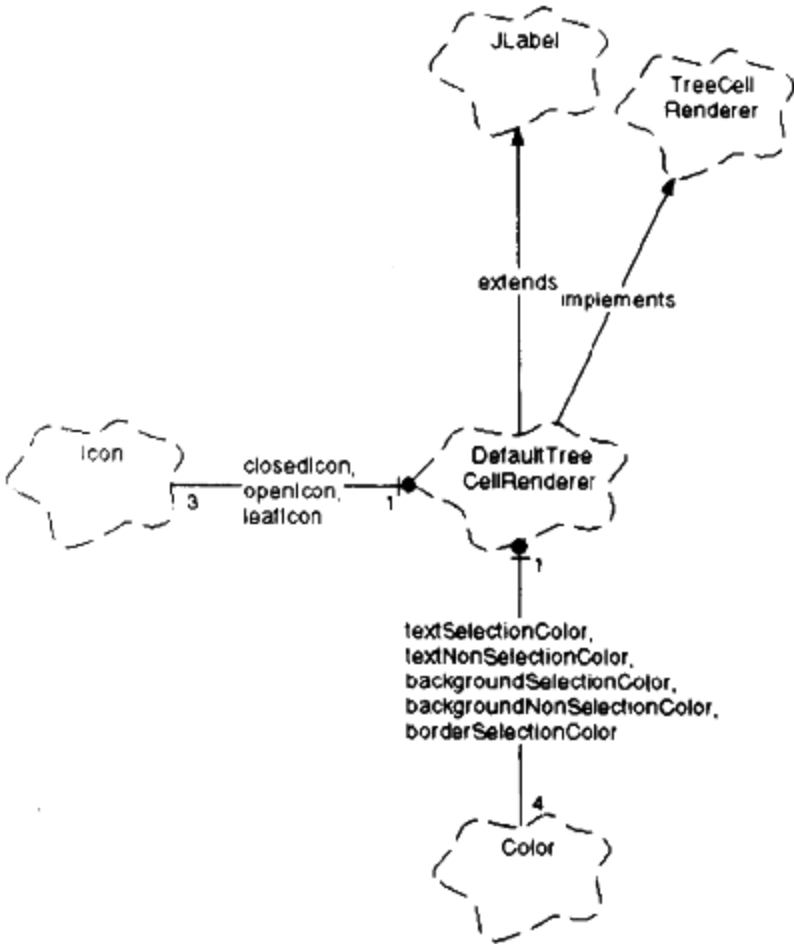


图 20-14 DefaultTreeCellRenderer 的类图

水平对齐方式设置为 `JLabel.LEFT`，并从 `UIManager` 类获取图标和颜色。

2. 方法

(1) 图标

```
public Icon getClosedIcon ( )
public Icon getLeafIcon ( )
public Icon getOpenIcon ( )
public void setClosedIcon (Icon)
public void setLeafIcon (Icon)
public void setOpenIcon (Icon)
public Icon getDefaultClosedIcon ( )
public Icon getDefaultLeafIcon ( )
public Icon getDefaultOpenIcon ( )
```

上面列出的这些方法是绘制器的树叶图标、打开图标和关闭图标的访问方法。

由于 `SwingL1FCS` 的一个错误，上面所列的最后三种方法返回的是绘制器的当前图标，而不是缺省图标。

(2) 颜色和字体

```
public void setBackground (Color)
public void setBackgroundNonSelectionColor (Color)
public void setBackgroundSelectionColor (Color)
public void setBorderSelectionColor (Color)
public Color getBackgroundNonSelectionColor ( )
```

```
public Color getBackgroundSelectionColor ( )
public Color getBorderSelectionColor ( )
public void setTextNonSelectionColor (Color)
public void setTextSelectionColor (Color)
public Color getTextNonSelectionColor ( )
public Color getTextSelectionColor ( )
public void setFont (Font)
```

上面所列的这些方法提供的是绘制器的颜色和字体的简单访问方法。`setBackground` 和 `setFont` 两个方法对 `JLabel` 类中同名方法进行了重载，如果颜色和字体不是 UI 资源，那么这两种方法分别只接受颜色和字体。有关 UI 资源的详细信息请参见 7.1.4 节“UI 资源”。

(3) 绘制/首选尺寸/绘制器组件

```
public Component getTreeCellRendererComponent (JTree, Object node,
    boolean selected, boolean expanded,
    boolean leaf, int row, boolean has Focus)
public Dimension getPreferredSize ( )
public void paint (Graphics)
```

`getTreeCellRendererComponent` 方法使用 `JTree.convertValueToText` 方法（只返回 `value.toString ()`）将传送给它的值转换为字符串。如果该值是 `DefaultMutableTreeNode` 的一个实例（绝大部分情况是如此），则 `DefaultMutableTreeNode.toString` 方法就返回代表该节点用户对象的字符串。用这个字符串来设置一个绘制器的文本（通过调用 `JLabel.setText ()`），而且这个绘制器的前景色和图标也被设置。

`DefaultTreeCellRenderer` 重载了 `JLabel` 类的 `paint` 方法，以便根据选取来填充背景。如果被绘制的节点有焦点，则 `paint` 方法还要绘制一个焦点边框。`paint` 方法随后调用 `super.paint ()`。

DefaultTreeCellRenderer 重载了 `getPreferredSize()` 方法，以便返回一个首选尺寸。这个尺寸比它的超类首选尺寸稍宽。

注意 关于 DefaultTreeCellRenderer.java 的 Javadoc 文档中说，DefaultTreeCellRenderer 实例的首选尺寸比绘制器超类的首选尺寸稍高，这是不对的。

1. 使用 DefaultTreeCellRenderer

可以用 DefaultTreeCellRenderer 来定制与树节点相关联的颜色、图标和字体。例如，图 20-15 所示的小应用程序就使用 DefaultTreeCellRenderer 的实例来修改树叶图标、打开图标、关闭图标和字体。

图 20-15 所示的小应用程序有一个简单的实现，其程序代码见例 20-8。

例 20-8 使用 DefaultTreeCellRenderer

```
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    static private Icon
        openFolder = new ImageIcon ("button_lit.jpg"),
        closedFolder = new ImageIcon ("button.jpg"),
        leafIcon = new ImageIcon ("leaf.gif");

    public void init () {
        JTree tree = new JTree ();
        JScrollPane scrollPane = new JScrollPane (tree);
        DefaultTreeCellRenderer renderer =
            new DefaultTreeCellRenderer ();

        renderer.setClosedIcon (closedFolder);
        renderer.setOpenIcon (openFolder);
        renderer.setLeafIcon (leafIcon);
        renderer.setFont (new Font ("Serif", Font.ITALIC, 12));

        tree.setCellRenderer (renderer);
        tree.setEditable (true);
        getContentPane ().add (scrollPane);
    }
}
```

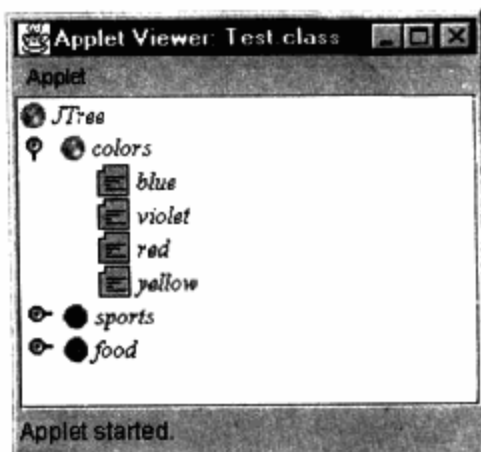


图 20-15 使用 DefaultTreeCellRenderer

这个小应用程序创建了 DefaultTreeCellRenderer 的一个实例，并使用 DefaultTreeCellRenderer 的方法来设置图标和字体。随后，把这个树的绘制器设置为新创建的绘制器。

利用 UIManager 类，可以设置所有树绘制器的图标和颜色。例 20-9 中列出的小应用程序和例 20-8 中的程序很相似，但在这个程序中，所有树的树叶图标、打开和关闭的图标都是由 UIManager 类设置的。有关 UIManager 类的详细信息请参见 7.1.3 节“UI 管理器”。

例 20-9 用 UIManager 类设置树图标的缺省值

```

import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    static private Icon
        openFolder = new ImageIcon ("button_lit.jpg"),
        closedFolder = new ImageIcon ("button.jpg"),
        leafIcon = new ImageIcon ("leaf.gif");

    public void init () {
        UIManager.put ("Tree.closedIcon", closedFolder);
        UIManager.put ("Tree.openIcon", openFolder);
        UIManager.put ("Tree.leafIcon", leafIcon);

        JTree tree = new JTree ();
        JScrollPane scrollPane = new JScrollPane (tree);
        getContentPane ().add (scrollPane);
    }
}

```

注意 例 20-9 中的小应用程序在创建树之前就利用 `UIManager.put` 方法指定了缺省图标。树只有在缺省值设置好以后创建，才可以采用新的缺省值。

2. 扩展 DefaultTreeCellRenderer

在绘制树的节点时，除了改变图标、颜色或字体外，有时还有必要修改节点的绘制方式。例如，图 20-16 所示的应用程序包含一个 `JTree` 文件查看器，这个查看器在文件夹之后增加了一个复选框，以模拟备份软件。这个复选框是由 `DefaultTreeCellRenderer` 的一个扩展来绘制的。

这个绘制器扩展 `DefaultTreeCellRenderer`，并实例化复选框、水平固定区和一个面板。固定区被用来在标签和复选框之间插入一些空格。这个面板是从绘制器的 `getTreeCellRendererComponent` 方法返回的。

这个绘制器的构造方法将这个面板的背景色设置为当前界面样式缺省的文本背景色，将绘制器、复选框和面板的不透明属性设置为 `false`。面板配备了一个 `FlowLayout` 实例，组件之间的水平和垂直空隙均为 0 像素。

这个绘制器（是一个标签，因为 `DefaultTreeCellRenderer` 扩展 `JLabel`）被添加到这个面板中。然后把水平固定区和复选框添加到这个面板中。

```

class FileNodeRenderer extends DefaultTreeCellRenderer {
    protected JCheckBox checkBox = new JCheckBox ("backup");
    private Component struct = Box.createHorizontalStrut (5);
    private JPanel panel = new JPanel ();

    public FileNodeRenderer () {

```

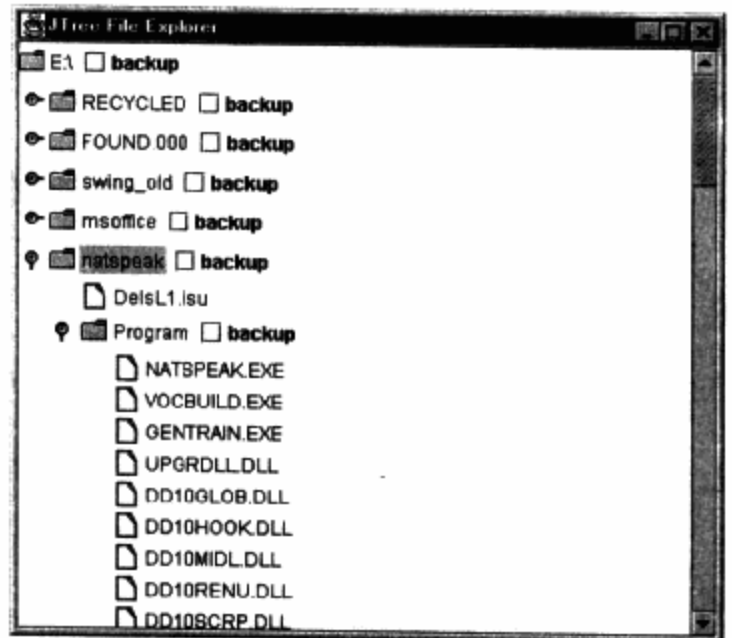


图 20-16 扩展 DefaultTreeCellRenderer


```

panel.setBackground (
    UIManager.getColor ("Tree.textBackground"));

setOpaque (false);
checkBox.setOpaque (false);
panel.setOpaque (false);

panel.setLayout (new FlowLayout (FlowLayout.CENTER, 0, 0));
panel.add (this);
panel.add (strut);
panel.add (checkBox);
}
...

```

这个绘制器的 `getTreeCellRendererComponent` 方法通过计算传送给它的值来获取对正绘制的节点的一个引用。该方法随后调用 `super.getTreeCellRendererComponent` 方法，这个方法配置绘制器的前景色和图标，如在类总结 20-3 中描述的那样。然后，根据节点是不是一个目录、有没有被选取来设置复选框的可见性属性和选取属性。由这个小应用程序模仿的简单备份软件只为目录而不是文件提供复选框。

包含一个标签（这个绘制器）、水平固定区和复选框的这个面板是从 `getTreeCellRendererComponent` 方法中返回的。

```

...
public Component getTreeCellRendererComponent (
    JTree tree, Object value,
    boolean selected, boolean expanded,
    boolean leaf, int row,
    boolean hasFocus) {
    FileNode node = (FileNode) value;

    super.getTreeCellRendererComponent (
        tree, value, selected, expanded,
        leaf, row, hasFocus);

    checkBox.setVisible (node.isDirectory ());
    checkBox.setSelected (node.isSelected ());

    return panel;
}
}

```

例 20-10 列出了图 20-16 所示的应用程序的代码。需要说明的是，例 20-10 列出的应用程序和前面例 20-4 列出的应用程序几乎完全相同，不同之处就在于这个程序的树配备了一个定制绘制器。因此，为简洁起见，例 20-10 没有列出这两个应用程序中重复的 `FileNode` 类部分。

例 20-10 扩展 `DefaultTreeCellRenderer`

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;
import java.io.File;

public class Test extends JFrame {
    public Test () {
        final JTree tree = new JTree (createTreeModel ());

```

```

JScrollPane scrollPane = new JScrollPane (tree);
FileNodeRenderer renderer = new FileNodeRenderer ();

tree.setEditable (true);
tree.setCellRenderer (renderer);

getContentPane ().add (scrollPane, BorderLayout.CENTER);

tree.addTreeExpansionListener (new TreeExpansionListener () {
    public void treeCollapsed (TreeExpansionEvent e) {
        // ...
    }
    public void treeExpanded (TreeExpansionEvent e) {
        TreePath path = e.getPath ();
        FileNode node = (FileNode)
            path.getLastPathComponent ();

        if ( ! node.isExplored () ) {
            DefaultTreeModel model =
                (DefaultTreeModel) tree.getModel ();

            node.explore ();
            model.nodeStructureChanged (node);
        }
    }
});

private DefaultTreeModel createTreeModel () {
    File root = new File ("E: /");
    FileNode rootNode = new FileNode (root), node;

    rootNode.explore ();
    return new DefaultTreeModel (rootNode);
}

public static void main (String args []) {
    GJApp.launch (new Test (), "JTree File Explorer",
        300, 300, 450, 400);
}

class FileNode extends DefaultMutableTreeNode {
    //Listing omitted: see Example 20-4 on page 1301
}

class FileNodeRenderer extends DefaultTreeCellRenderer {
    protected JCheckBox checkBox = new JCheckBox ("backup");
    private Component strut = Box.createHorizontalStrut (5);
    private JPanel panel = new JPanel ();

    public FileNodeRenderer () {
        panel.setBackground (
            UIManager.getColor ("Tree.textBackground"));

        setOpaque (false);
        checkBox.setOpaque (false);
        panel.setOpaque (false);

        panel.setLayout (new FlowLayout (FlowLayout.CENTER, 0, 0));
        panel.add (this);
        panel.add (strut);
        panel.add (checkBox);
    }
}

```

```

public Component getTreeCellRendererComponent (
    JTree tree, Object value,
    boolean selected, boolean expanded,
    boolean leaf, int row,
    boolean hasFocus) {
    FileNode node = (FileNode) value;

    super.getTreeCellRendererComponent (
        tree, value, selected, expanded,
        leaf, row, hasFocus);

    checkBox.setVisible (node.isDirectory ());
    checkBox.setSelected (node.isSelected ());

    return panel;
}

```

应该注意的是，不能操纵由定制绘制器绘制的复选框，因为绘制器使用像橡皮图章一样的组件来绘制单元。换句话说，实际上复选框并没有添加到这个树中，因此不能选取或取消选取。如果要操纵复选框，这个树必须配备一个定制的编辑器，20.8 节“绘制和编辑：学习一个样例”中说明了这一点。

3. 树格式化绘制器

有时需要根据某些标准来格式化单元值。例如，图 20-17 中所示的树将某些节点设为货币的格式。

利用 `DefaultTreeCellRenderer.setValue` 方法，Swing 的 table 单元绘制器可以很方便地实现格式化绘制器，这个方法在 19.6.4 节“表格格式化绘制器”一节中进行了描述。遗憾的是，树单元绘制器的实现形式是不一样的，因此与表格相比，开发人员在实现树的格式化绘制器时需要多做些工作。

例 20-11 列出了图 20-17 中所示的小应用程序的代码。这个小应用程序创建 `JTree` 的一个实例，并向其提供缺省的可变树节点。树的绘制器被设置为 `FormattingRenderer` 的一个实例，它格式化代表价格的节点。

例 20-11 把用户对象设置为货币格式

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
import java.text.*;

public class Test extends JApplet {
    public void init () {
        JTree tree = new JTree ();
        JScrollPane scrollPane = new JScrollPane (tree);

        DefaultMutableTreeNode root =
            new DefaultMutableTreeNode ("prices");

        root.add (new DefaultMutableTreeNode (new Double (10.99)));
        root.add (new DefaultMutableTreeNode (new Double (8.99)));
    }
}

```

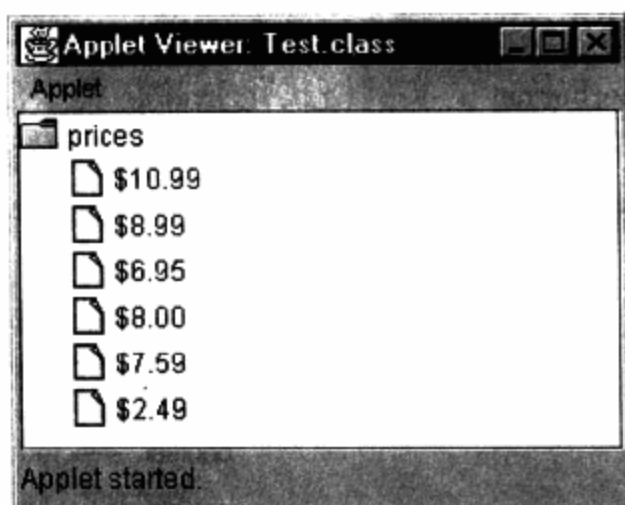


图 20-17 一个格式化的绘制器

```

root.add (new DefaultMutableTreeNode (new Double (6.95)));
root.add (new DefaultMutableTreeNode (new Double (8.00)));
root.add (new DefaultMutableTreeNode (new Double (7.59)));
root.add (new DefaultMutableTreeNode (new Double (2.49)));

DefaultTreeModel model =
    (DefaultTreeModel) tree.getModel ();

model.setRoot (root);

tree.setCellRenderer (new FormattingRenderer ());
getContentPane ().add (scrollPane);

```

这个格式化绘制器扩展 DefaultTreeCellRenderer，而且，与本章在“扩展 DefaultTreeCellRenderer”一节中讨论过的绘制器一样，格式化绘制器调用 super.TreeCellRenderer Component () 来初始化绘制器的前景色和图标，获取与正被绘制的节点相关联的用户对象的一个引用，用 NumberFormat 的实例格式化 Double 类型的用户对象，NumberFormat 在 Java.text 包中。然后，把格式化后的字符串设置为这个绘制器的文本。最后，这个绘制器从 getTreeCellRendererComponent 方法中返回其本身。

```

...
class FormattingRenderer extends DefaultTreeCellRenderer {
    public Component getTreeCellRendererComponent (
        JTree tree, Object value,
        boolean selected, boolean expanded,
        boolean leaf, int row,
        boolean hasFocus) {
        // initialize renderer component (this) ...
        super.getTreeCellRendererComponent (
            tree, value, selected, expanded,
            leaf, row, hasFocus);

        //now format label text...
        DefaultMutableTreeNode n = (DefaultMutableTreeNode) value;
        Object userObject = n.getUserObject ();
        if (userObject instanceof Double) {
            Double d = (Double) userObject;
            Format format = NumberFormat.getCurrencyInstance ();
            setText (value == null ? "" : format.format (d));
        }
        return this;
    }
}

```

Swing 提示

树格式化绘制器与表格格式化绘制器的比较

DefaultTableCellRenderer 类和 DefaultTreeCellRenderer 类都扩展 JLabel，而且都有助于显示文字和（或）图标。DefaultTableCellRenderer 实现了一个 protected 的 setValue 方法，该方法在更新了绘制器的边框、颜色和字体以后被调用。DefaultTableCellRenderer 的扩展可以重载 setValue 方

法，从而很容易地实现格式化绘制器。

与 DefaultTableCellRenderer 不同，DefaultTreeCellRenderer 类不实现 setValue 方法。因此，树格式化绘制器通常重载 getTreeCellRendererComponent，并调用 super.getTreeCellRendererComponent() 来更新绘制器的边框、颜色和字体。

20.6.2 Metal 界面样式

如果为一个树指定了带有关键字 JTree.lineStyle 的客户属性，那么在树的节点之间绘制连线的样式就会受到 Metal 界面样式树的影响。可以和 JTree.lineStyle 一起使用的客户属性有三个：

- "None"
- "Horizontal"
- "Angled"

图 20-18 所示的应用程序展示了对应于上面所列的三个字符串的连线情况。



图 20-18 Metal 界面样式的 JTree.lineStyle 客户属性

JTree.lineStyle 客户属性按如下方式设置：

```
tree.setClientProperty ("JTree.lineStyle", "Angled");
```

20.6.3 根节点和根句柄

JTree 的实例可以设置它们根节点和根节点句柄的可见性。句柄是文件夹左侧的那些小控件，它用于展开和折叠节点。

图 20-19 所示的小应用程序包含一个树和两个复选框。这两个复选框用来控制根节点和根节点句柄的可见性。

这个小应用程序的两个复选框都添加了动作监听器，以调用相应的 JTree 的访问方法，来访问根节点和根节点句柄的可见性。例 20-12 列出了图 20-19 所示的小应用程序的代

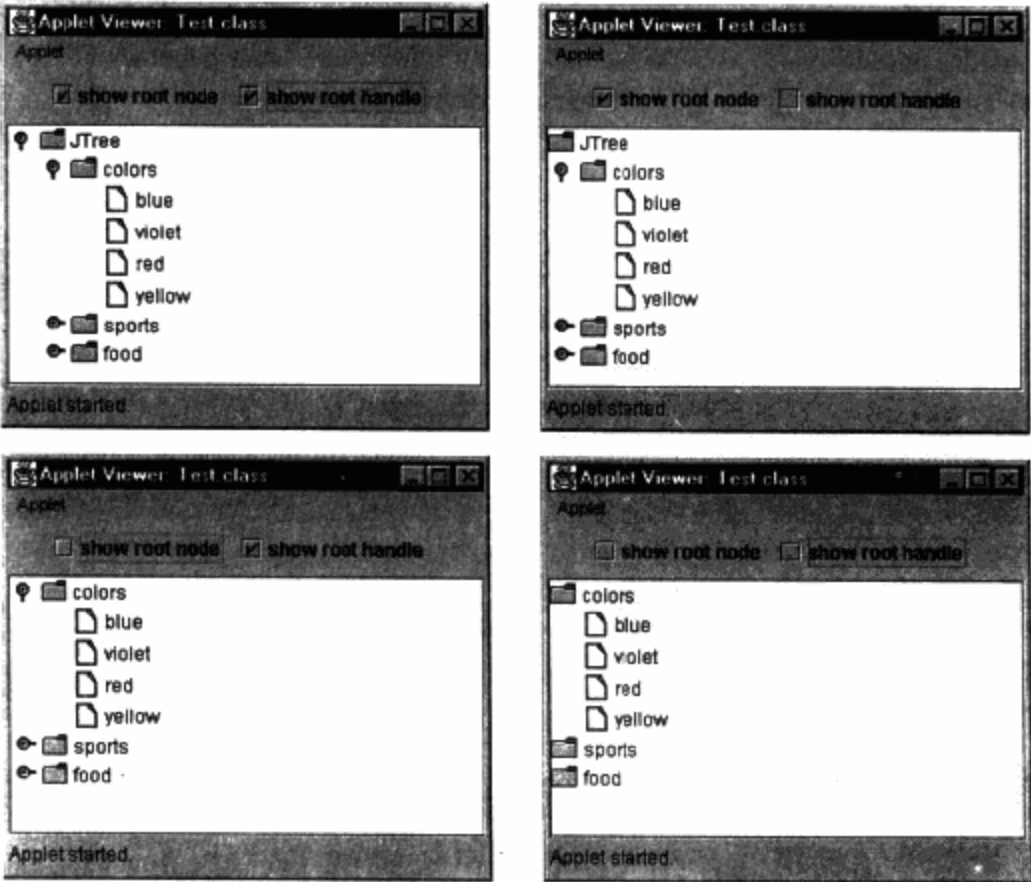


图 20-19 显示根节点和根节点句柄

码。

例 20-12 显示根节点和根节点句柄

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    JTree tree = new JTree ();

    public void init () {
        Container contentPane = getContentPane ();
        JScrollPane scrollPane = new JScrollPane (tree);

        contentPane.add (new ControlPanel (), BorderLayout.NORTH);
        contentPane.add (scrollPane, BorderLayout.CENTER);
    }

    class ControlPanel extends JPanel {
        JCheckBox showRoot = new JCheckBox ("show root node");
        JCheckBox showRootHandles = new JCheckBox (
            "show root handle");

        public ControlPanel () {
            initializeCheckBoxes ();
            setLayout (new FlowLayout ());
            add (showRoot);
            add (showRootHandles);

            showRoot.addActionListener (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    tree.setRootVisible (showRoot.isSelected ());
                }
            });

            showRootHandles.addActionListener (
                new ActionListener () {
                    public void actionPerformed (ActionEvent e) {
                        tree.setShowsRootHandles (
                            showRootHandles.isSelected ());
                    }
                }
            );
        }

        private void initializeCheckBoxes () {
            showRoot.setSelected (tree.isRootVisible ());
            showRootHandles.setSelected (
                tree.getShowsRootHandles ());
        }
    }
}
```

20.7 树单元编辑

树单元编辑器是由 `TreeCellEditor` 接口定义的, `TreeCellEditor` 接口扩展 `CellEditor` 接口。下面有关树单元编辑的介绍假定读者已经理解了 `CellEditor` 接口 (“单元编辑器”一节对 `CellEditor` 接口进行了讨论)。

接口总结 20-6 对 TreeCellEditor 接口进行了总结。

接口总结 20-6 TreeCellEditor

扩展: CellEditor

```
public abstract Component getTreeCellEdit orComponent (JTree tree,  
                                                         Object value,  
                                                         boolean isSelected,  
                                                         boolean expanded,  
                                                         boolean leaf,  
                                                         int row)
```

除了由 CellEditor 接口定义的方法之外, TreeCellEditor 接口还定义了 getTreeCellEdit orComponent 方法, 该方法返回一个组件。

树单元编辑器除实现由 CellEditor 接口定义的方法以外, 还必须提供一个用于编辑的组件。对绘制器来说, 情况经常是: 从 TreeCellEditor.getTreeCellEditorComponent () 方法返回的组件就是编辑器本身。

Swing 提供了 TreeCellEditor 接口的两种实现: DefaultCellEditor 类和 DefaultTreeCellEditor 类。DefaultCellEditor 的实例既可用于树也可用于表, 因为 DefaultCellEditor 实现 TableCellEditor 接口。

DefaultTreeCellEditor 类是一个树单元编辑器, 它在编辑时并不除去节点的图标。在 20.7.2 节“使用 DefaultTreeCellEditor”中对 DefaultTreeCellEditor 类进行了讨论。

20.7.1 扩展 DefaultCellEditor

图 20-20 所示的树有一个编辑器。该编辑器扩展 DefaultCellEditor 类, 它提供一个带一个颜色列表的组合框。只有那些父节点是 colors 节点的节点才可以激活这个编辑器。图 20-20 从左至右显示了一个由蓝色变为黄色的节点。

例 20-13 列出了图 20-20 所示的小应用程序的代码。这个小应用程序创建 ColorEditor 的一个实例, 指定该实例为树的编辑器。这个小应用程序调用 JTree.setEditable (true), 这是因为在缺省情况下 JTree 的实例是不可编辑的。

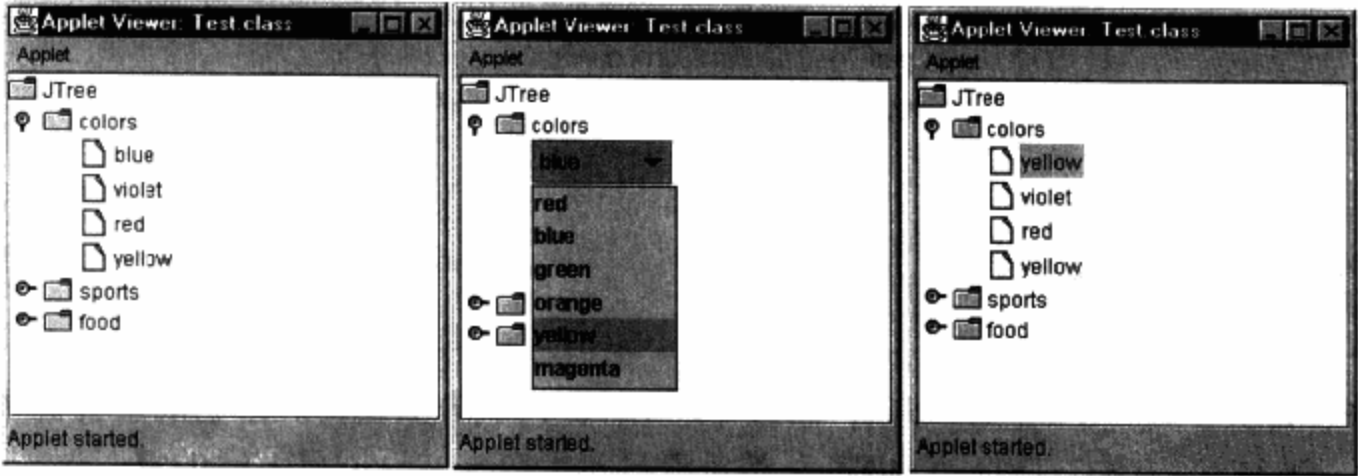


图 20-20 扩展 DefaultCellEditor

例 20-13 一个扩展 DefaultCellEditor 的编辑器

```
import javax.swing.*.*;  
import javax.swing.tree.*.*;  
import java.awt.*.*;  
import java.awt.event.*.*;
```



```

import java.util.*;

public class Test extends JApplet {
    public void init () {
        JTree tree = new JTree ();
        JScrollPane scrollPane = new JScrollPane (tree);
        JComboBox combo = new JComboBox ();

        combo.addItem ("red");
        combo.addItem ("blue");
        combo.addItem ("green");
        combo.addItem ("orange");
        combo.addItem ("yellow");
        combo.addItem ("magenta");

        tree.setCellEditor (new ColorEditor (tree, combo));
        tree.setEditable (true);

        getContentPane ().add (scrollPane);
    }
}

class ColorEditor extends DefaultCellEditor {
    private JTree tree;

    public ColorEditor (JTree tree, JComboBox comboBox) {
        super (comboBox);
        this.tree = tree;
    }

    public boolean isCellEditable (EventObject e) {
        boolean rv = false; // return value

        if (e instanceof MouseEvent) {
            MouseEvent me = (MouseEvent) e;

            if (me.getClickCount () == 3) {
                TreePath path =
                    tree.getPathForLocation (me.getX (), me.getY ());

                if (path.getPathCount () == 1) // root node
                    return false;

                DefaultMutableTreeNode node =
                    (DefaultMutableTreeNode)
                        path.getLastPathComponent ();
                rv = node.getParent ().toString ().equals ("colors");
            }
        }

        return rv;
    }
}

```

ColorEditor 类扩展 DefaultCellEditor，而且 ColorEditor 构造方法将传送给它的组合框再传送给 DefaultCellEditor 构造方法。DefaultCellEditor 类负责显示组合框并返回已编辑的值。

因为只有表示颜色的节点才可以使用颜色编辑器，所以 ColorEditor 类重载 CellEditor 接口的 isCellEditable () 方法。对父节点是 colors 文件夹的节点，这个方法就返回 true，对所有其他的节点就返回 false。三击鼠标，颜色编辑器就会被激活。

20.7.2 DefaultTreeCellEditor

DefaultTreeCellEditor 类是后来添加到 Swing1.1 中的。添加的必要性在于，DefaultCellEditor

的实例总趋向于去除节点的图标，从图 20-20 中可以看到这一点。

DefaultTreeCellEditor 的实例是“真正”编辑器的封套。把这个真正的编辑器放置在一个容器中，当激活这个编辑器时，这个容器就添加到树中。DefaultTreeCellEditor 类还维护对 DefaultTreeCellEditor 实例的一个引用，这个编辑器使用这个实例来获取一个在它旁边被绘制的图标。

三击，或者双击鼠标，然后延迟 1200 毫秒，就会激活缺省的树单元编辑器。

DefaultTreeCellEditor 类的类图见图 20-21。



图 20-21 DefaultTreeCellEditor 的类图

DefaultTreeCellEditor 扩展 Object 类，并实现 TreeCellEditor 接口。DefaultTreeCellEditor 类维护对它的真正编辑器和缺省树单元绘制器的引用，树单元绘制器用来获取节点的图标。

DefaultTreeCellEditor 还维护许多对其他对象的引用，如真正编辑器所在的编辑容器、从真正编辑器获取的编辑组件和从绘制器获取的编辑图标等。

DefaultTreeCellEditor 还维护许多属性，包括用于放置编辑容器距离节点左边缘的偏移量和用于跟踪 1200 毫秒延迟的计时器。

DefaultTreeCellEditor 维护对树和传送给 getTreeCellEditorComponent 方法的最后一行的引用。DefaultTreeCellEditor 的实例维护对编辑器所使用字体的引用。

类总结 20-4 对 DefaultTreeCellEditor 进行了总结。

类总结 20-4 DefaultTreeCellEditor

扩展：Object

实现：TreeCellEditor

1. 构造方法：

```
public DefaultTreeCellEditor (JTree, DefaultTreeCellRenderer)
public DefaultTreeCellEditor (JTree, DefaultTreeCellRenderer, TreeCellEditor)
```

DefaultTreeCellEditor 提供了两个构造方法。这两个构造方法都需要向其传送编辑器使用的树和 DefaultTreeCellRenderer 的一个实例，这个实例用于获取编辑图标。上面列出的第二个构造方法还需要一个树单元编辑器。

如果在构造时没有指定树单元编辑器，则将使用一个包含文本字段的缺省编辑器。

2. 方法

(1) CellEditor 方法

```
public void addCellEditorListener (CellEditorListener)
public void removeCellEditorListener (CellEditorListener)

public void cancelCellEditing ()
public boolean stopCellEditing ()

public boolean isCellEditable (EventObject)
public boolean shouldSelectCell (EventObject)
public Object getCellEditorValue ()
```

上面列出的这些方法都是由 CellEditor 接口定义的，除 isCellEditable () 外的其他方法都交给真正编辑器。如果传送给 isCellEditable () 的事件代表三击鼠标，或者第二次单击后已经延迟了 1200 毫秒，则 isCellEditable () 将返回 true。

(2) TreeCellEditor 方法

```
public abstract Component getTreeCellEditorComponent (JTree tree,
                                                         Object value,
                                                         boolean isSelected,
                                                         boolean expanded,
                                                         boolean leaf,
                                                         int row)
```

getTreeCellEditorComponent 方法从真正编辑器那里获取这个编辑器组件并配置编辑容器的字体。编辑容器是从 getTreeCellEditorComponent 方法返回的。

(3) 开始编辑

```
protected boolean shouldStartEditingTimer (EventObject)
protected void startEditingTimer ()
protected boolean canEditImmediately (EventObject)
protected void actionPerformed (ActionEvent)
```

上面列出的各方法协同一致，在单击鼠标/暂停/再单击并延迟 1200 毫秒之后激活编辑器。在 1200 毫秒的延迟过后编辑器的计时器就调用 actionPerformed 方法。

(4) 边框选取颜色/字体/树属性

```
public Color getBorderSelectionColor ()
public Font getFont ()
public void setBorderSelectionColor (Color)
public void setFont (Font)
protected void setTree (JTree)
```

DefaultTreeCellEditor 提供对它的字体、树和边框选取颜色的访问方法。

(5) 实用方法

```
protected Container createContainer ()
protected TreeCellEditor createTreeCellEditor ()
protected void determineOffset (JTree tree,
                                Object value,
```

```

boolean isSelected,
boolean expanded,
boolean leaf,
int row)

```

```

protected boolean inHitRegion (int, int)
protected void prepareForEditing ()

```

上面列出的方法是 DefaultTreeCellEditor 类内部使用的实用方法。这些方法是 protected，因此 DefaultTreeCellEditor 的扩展也可以使用它们。

(6) TreeSelectionListener 方法

```
public void valueChanged (TreeSelectionEvent)
```

DefaultTreeCellEditor 的实例将其自身登记为树的树选取监听器，以便探测 1200 毫秒延迟时间内的选取情况。在延迟时间内如果发生了选取，就不会进行编辑。

使用 DefaultTreeCellEditor

图 20-22 中所示的小应用程序和前面图 20-20 所示的小应用程序很相似，只是在这个程序中树配备有一个 DefaultTreeCellEditor 实例。因为这个树配备 DefaultTreeCellEditor 的一个实例而不是 DefaultCellEditor 的实例，所以把这个编辑器放置于节点图标右边。

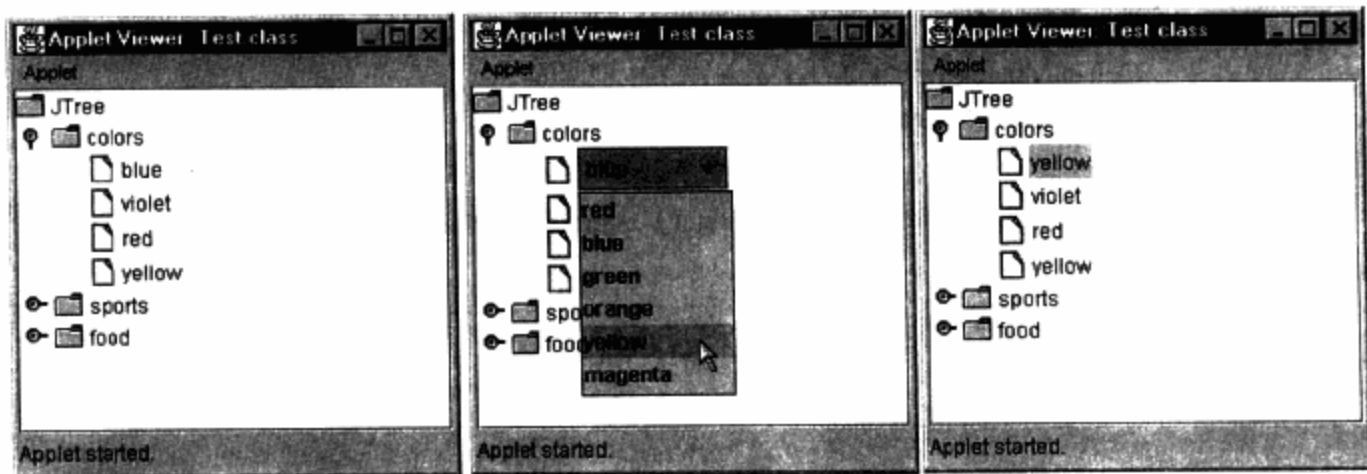


图 20-22 使用 DefaultTreeCellEditor

例 20-14 列出了图 20-22 所示小应用程序的程序代码。

例 20-14 使用 DefaultTreeCellEditor

```

import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    public void init () {
        JTree tree = new JTree ();
        JScrollPane scrollPane = new JScrollPane (tree);
        JComboBox combo = new JComboBox ();

        combo.addItem ("red");
        combo.addItem ("blue");
        combo.addItem ("green");
        combo.addItem ("orange");
        combo.addItem ("yellow");
        combo.addItem ("magenta");
    }
}

```

```

        tree.setCellEditor (new DefaultTreeCellEditor (
            tree, new DefaultTreeCellRenderer (),
            new ColorEditor (tree, combo));
        tree.setEditable (true);
        getContentPane ().add (scrollPane);
    }
}

class ColorEditor extends DefaultCellEditor {
    private JTree tree;

    public ColorEditor (JTree tree, JComboBox comboBox) {
        super (comboBox);
        this.tree = tree;
    }

    public boolean isCellEditable (EventObject e) {
        boolean rv = false; // return value
        if (e instanceof MouseEvent) {
            MouseEvent me = (MouseEvent) e;
            if (me.getClickCount () == 3) {
                TreePath path =
                    tree.getPathForLocation (me.getX (), me.getY ());
                DefaultMutableTreeNode node =
                    (DefaultMutableTreeNode)
                        path.getLastPathComponent ();
                rv = node.getParent ().toString ().equals ("colors");
            }
        }
        return rv;
    }
}

```

要说明的是,例 20-14 列出的小应用程序和例 20-13 中列出的小应用程序是一样的,只是在这里把 ColorEditor 封装在 DefaultTreeCellEditor 的一个实例中。

20.8 绘制和编辑: 学习一个样例

前面关于树单元绘制和编辑的例子都很简单,本节讨论一个复杂些的例子。

图 20-23 所示的应用程序模拟备份软件,备份那些备份复选框被选中的目录。为方便说明,这里只有文件夹才具有备份复选框。

图 20-23 所示的应用程序是本章“扩展 DefaultMutableTreeNode”小节讨论过的小应用程序和“扩展 DefaultTreeCellEditor”一节讨论过的应用程序的综合。和前一个应用程序一样,图 20-23 中所示的树包含 FileNode 类的一些实例,这些实例允许查

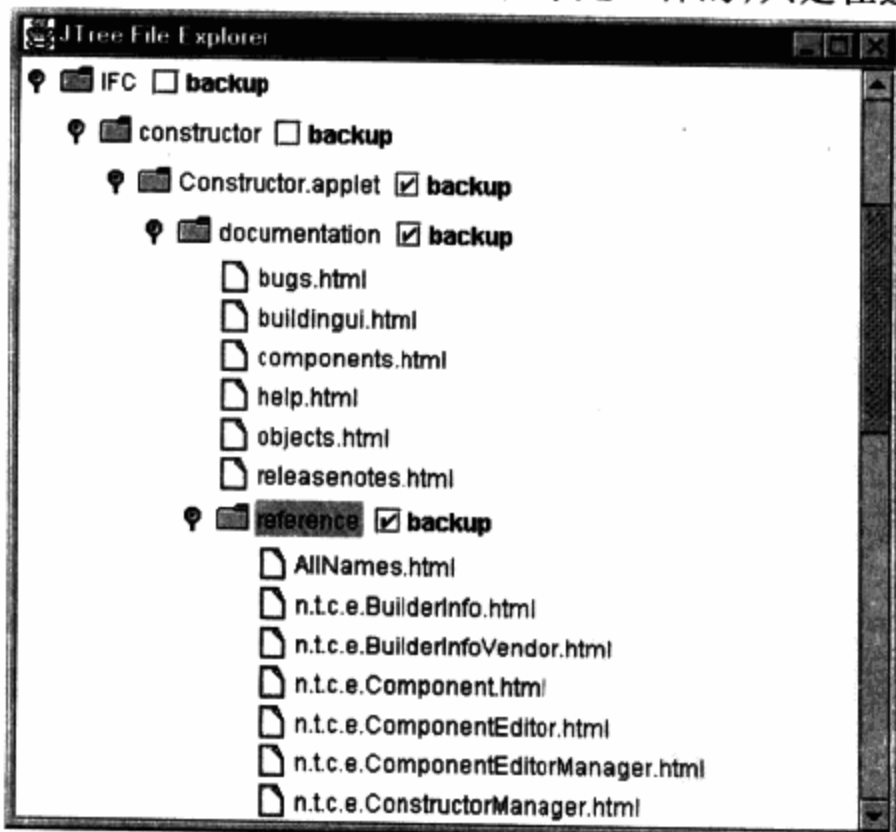


图 20-23 高级绘制和编辑

看目录层次。此外，和后一个应用程序相同的是，图 20-23 中的树配备了一个定制绘制器来绘制复选框。

与这两个应用程序不同的是，图 20-23 中的树有一个定制编辑器，允许对复选框进行选取和取消选取等操作。

图 20-23 所示的应用程序实现了很多类，这些类将在随后的各节中予以讨论。在对这些类逐个讨论之前，介绍一下这些类和它们的功能可能是有益的。下面按类出现的顺序列出了各个类。

Test——扩展 `JFrame` 并创建树及与其相关联的绘制器和编辑器。该类还向树添加了展开和编辑监听器。

SelectableFile——一个 `File` 实例的简单封套。可以选取和取消选取 `SelectableFile` 的实例，其选取状态和与文件夹相关联的复选框保持一致。

FileNode——`DefaultMutableTreeNode` 的一个扩展，它维护 `SelectableFile` 的一个实例，并把它作为其用户对象。

FileNodeRenderer——`DefaultTreeCellEditor` 的一个扩展，它绘制一个 `FileNode` 的文件名。如果文件节点代表的是文件夹，则这个类还绘制复选框（可选）。

FileNode Editor Renderer——树编辑器使用的 `FileNodeRenderer` 的一个扩展。

FileNodeEditor——一个简单的编辑器，它使用 `FileNodeRenderer` 的一个实例来产生这个编辑器的组件。

ImmediateEditor——`DefaultTreeCellEditor` 的一个扩展，当鼠标在绘制器的复选框中单击时，它使鼠标单击的编辑生效。这个编辑器还确定一个事件是否应该选取一个节点。

上面列出的各类中有些在前面已经讨论过了，但这里还有必要旧话重提。`ImmediateEditor` 类和 `FileNodeEditor` 类说明了树单元编辑更高级的功能。

回想一下，在本章“`DefaultTreeCellEditor`”一节中，`DefaultTreeCellEditor` 的实例用作“真正”编辑器的封套。`ImmediateEditor` 类扩展 `DefaultTreeCellEditor`，`ImmediateEditor` 的真正编辑器是一个 `FileNodeEditor`。

`ImmediateEditor` 重载 `DefaultTreeCellEditor.canEditImmediately()` 和 `DefaultTreeCellEditor.shouldSelectCell()`。重载前者是为了在单击鼠标后使编辑生效，以取代 `DefaultTreeCellEditor` 提供的缺省方式：三击鼠标，或单击/暂停/单击并延迟 1200 毫秒后才可编辑。重载 `shouldSelectCell` 方法是为了在有鼠标单击文件夹复选框的情况下禁止选取。

注意 由于在 `JDK1.2` 中有一个 `AWT` 事件处理错误，所以编辑节点的复选框需要单击鼠标两次。

20.8.1 Test 类

图 20-23 中的应用程序用树模型创建了 `JTree` 的一个实例，这个树模型的根节点是 `FileNode` 的一个实例，代表 E 驱动器。

该应用程序还创建了 `FileNodeRenderer` 和 `FileNodeEditor` 的实例。把 `FileNodeEditor` 实例传送给 `ImmediateEditor` 构造方法，从而指定为树的真正编辑器。`ImmediateEditor` 构造方法再将 `FileNodeEditor` 实例传送给 `DefaultTreeCellEditor` 构造方法。

`Test` 构造方法调用 `JTree.setEditable(true)`，以便允许编辑这个树。

树中添加了两个监听器，以处理展开和编辑事件。通过查看未查看过的节点来处理展开事件，通过显示这个事件的有关信息来处理编辑事件。

例 20-15 绘制和编辑：一个学习样例

```
import javax.swing.*.*;
```

[illegible]


```

private DefaultTreeModel createTreeModel () {
    File root = new File ("E: /");
    FileNode rootNode = new FileNode (root);

    rootNode.explore ();
    return new DefaultTreeModel (rootNode);
}

public static void main (String args []) {
    GJApp.launch (new Test (), "JTree File Explorer",
                  300, 300, 450, 400);
}

// End of Test class
...

```

20.8.2 SelectableFile 类和 FileNode 类

图 20-23 显示的树含有 FileNode 类型的节点，这些节点把 SelectableFile 的实例作为它们的用户对象。SelectableFile 类把一个 boolean selected 属性添加到一个文件中。

```

...
class SelectableFile {
    private File file;
    private boolean selected = false;
    public SelectableFile (File file) {
        this.file = file;
    }
    public String toString () {
        return file.toString () + " selected: " + selected;
    }
    public void setSelected (boolean s) { selected = s; }
    public boolean isSelected () { return selected; }
    public File getFile () { return file; }
}
...

```

FileNode 类扩展 DefaultMutableTreeNode，并把它的用户对象设置为 SelectableFile 的一个实例。代表目录的 FileNode 是文件夹，而文件是树叶。FileNode 提供对它的用户对象的选取状态的访问方法，允许把 FileNode 实例作为可选取的对象来操纵。FileNode 类可以查看子文件和文件夹，而且还可以跟踪这个节点是否已查看过。

```

...
class FileNode extends DefaultMutableTreeNode {
    private boolean explored = false;

    public FileNode (File file) {
        setUserObject (new SelectableFile (file));
    }

    public boolean getAllowsChildren () { return isDirectory (); }
    public boolean isLeaf () { return ! isDirectory (); }

    public File getFile () {
        SelectableFile sf = (SelectableFile) getUserObject ();
        return sf.getFile ();
    }

    public boolean isSelected () {

```

```

        SelectableFile sf = (SelectableFile) getUserObject ();
        return sf.isSelected ();
    }

    public void setSelected (boolean b) {
        SelectableFile sf = (SelectableFile) getUserObject ();
        sf.setSelected (b);
    }

    public boolean isDirectory () {
        File file = getFile ();
        return file.isDirectory ();
    }

    public String toString () {
        File file = getFile ();
        String filename = file.toString ();
        int index = filename.lastIndexOf ("\\");

        return (index != -1 && index != filename.length () - 1) ?
            filename.substring (index + 1) :
            filename;
    }

    public void explore () { explore (); }
    public boolean isExplored () { return explored; }

    public void explore () {
        if (! isExplored ()) {
            File file = getFile ();
            File [] children = file.listFiles ();

            for (int i=0; i < children.length; ++ i)
                add (new FileNode (children [i]));

            explored = true;
        }
    }
    ...

```

20.8.3 绘制器

图 20-23 示出的应用程序使用了两个绘制器：一个用于绘制树中的节点，另一个用于编辑器。由于绘制和编辑使用的组件是相同的，因此编辑器的绘制器（FileNodeEditorRenderer）是树的绘制器（FileNodeRenderer）的一个扩展。

1. FileNodeRenderer

FileNodeRenderer 类与 20.2.3 节介绍的同名类相似，只是添加了一个方法，这个方法计算复选框距绘制器面板左边界的偏移量。该方法名为 getCheckBoxOffset，供 ImmediateEditor 类使用，该类用这个偏移量来确定当鼠标在文件夹中单击时是否应该选取这个文件夹或激活这个编辑器。

注意 所有用 FileNodeRenderer 绘制的节点都有复选框，但只对文件夹是可见的。

```

...
class FileNodeRenderer extends DefaultTreeCellRenderer {
    protected JCheckBox checkBox = new JCheckBox ("backup");
    private Component strut = Box.createHorizontalStrut (5);
    private JPanel panel = new JPanel ();

```

```

public FileNodeRenderer () {
    panel.setBackground (
        UIManager.getColor ("Tree.textBackground"));

    setOpaque (false);
    checkBox.setOpaque (false);
    panel.setOpaque (false);

    panel.setLayout (new FlowLayout (FlowLayout.CENTER, 0, 0));
    panel.add (this);
    panel.add (strut);
    panel.add (checkBox);
}

public Component getTreeCellRendererComponent (
    JTree tree, Object value,
    boolean selected, boolean expanded,
    boolean leaf, int row,
    boolean hasFocus) {
    FileNode node = (FileNode) value;

    super.getTreeCellRendererComponent (
        tree, value, selected, expanded,
        leaf, row, hasFocus);

    checkBox.setVisible (node.isDirectory ());
    checkBox.setSelected (node.isSelected ());
    return panel;
}

public Dimension getCheckBoxOffset () {
    Graphics g = panel.getGraphics ();
    int xoffset = 0;

    if (g != null) {
        try {
            FontMetrics fm = g.getFontMetrics ();
            xoffset = fm.stringWidth (getText ()) +
                strut.getPreferredSize ().width;
        }
        finally {
            g.dispose ();
        }
    }

    return new Dimension (xoffset, 0);
}

...

```

`getCheckBoxOffset` 还可以由编辑器使用，用于确定在鼠标单击某个节点时是否应该选取该节点。

2. FileNodeEditorRenderer

`FileNodeEditorRenderer` 的实例由 `FileNodeEditor` 使用，以提供编辑组件。`FileNodeEditorRenderer` 类扩展 `FileNodeRenderer`，即用同一个组件来绘制节点和编辑节点。然而我们知道，`ImmediateEditor` (`DefaultTreeCellEditor` 的一个扩展) 把真正的编辑器 (这里指 `FileNode` 编辑器的一个实例) 放在这个文件夹图标的右边。结果，`FileNodeEditorRenderer` 将其图标设置为 `null`，以免在安装编辑器组件时这两个图标被画在一起。

```

...
class FileNodeEditorRenderer extends FileNodeRenderer {
    public Component getTreeCellRendererComponent (
        JTree tree, Object value,
        boolean selected, boolean expanded,
        boolean leaf, int row,
        boolean hasFocus) {
        Component c = super.getTreeCellRendererComponent (tree,
            value, selected, expanded,
            leaf, row, hasFocus);

        setIcon (null);
        return c;
    }
    public JCheckBox getCheckBox () {
        return checkBox;
    }
}
...

```

20.8.4 编辑器

图 20-3 中示出的应用程序实现了两个编辑器：ImmediateEditor 类和 FileNodeEditor 类。

ImmediateEditor 类是 DefaultTreeCellEditor 的一个扩展，它被指定为树的编辑器。

FileNodeEditor 类代表传送给 ImmediateEditor 构造方法的真正编辑器。

1. FileNodeEditor

FileNodeEditor 类是 AbstractCellEditor 的一个简单扩展，它使用一个 FileNodeEditorRenderer 实例来返回这个编辑器的组件。注意，AbstractCellEditor 类不是 Swing 类，但是它是 CellEditor 接口的一个有用的实现，详细内容请参见 19.6.7 节中的“AbstractCellEditor”。

FileNodeEditor 类创建 FileNodeEditorRenderer 的一个实例，并在它的构造方法中创建了 JCheckBox。

要为复选框添加一个动作监听器，以便设置编辑节点的选取状态和调用 stopCellEditing ()，该方法结束这次编辑会话。因此，鼠标单击负责开始和停止编辑。

FileNode 重载 getCellEditorValue () 来返回编辑节点的用户对象，该对象是 SelectableFile 的一个实例。

```

class FileNodeEditor extends AbstractCellEditor {
    FileNodeEditorRenderer renderer;
    FileNode nodeBeingEdited;
    JCheckBox checkBox;

    public FileNodeEditor () {
        renderer = new FileNodeEditorRenderer ();
        checkBox = renderer.getCheckBox ();

        checkBox.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                nodeBeingEdited.setSelected (checkBox.isSelected ());
                stopCellEditing ();
            }
        });
    }

    public Component getTreeCellEditorComponent (
        JTree tree, Object value,

```

```

        boolean selected, boolean expanded,
        boolean leaf, int row) {
    nodeBeingEdited = (FileNode) value;

    return renderer.getTreeCellRendererComponent (tree,
        value, selected, expanded,
        leaf, row, true); // hasFocus ignored
}

public Object getCellEditorValue () {
    return nodeBeingEdited.getUserObject ();
}
}

```

2. ImmediateEditor

ImmediateEditor 类是 DefaultTreeCellEditor 的一个扩展，它有两个作用：一个作用是当在一个节点中检测到鼠标单击时就立即启动编辑，另一个作用是当选中复选框时就禁止选取。

缺省情况下，DefaultTreeCellEditor 在三次单击或单击/暂停/单击并延迟 1200 毫秒之后启动编辑。对图 20-23 所示的应用程序而言，更希望在节点中检测到鼠标单击后立即启动编辑。由于 ImmediateEditor 类可以立即启动编辑，所以在用户看来，由树绘制器绘制的复选框实际上已经添加到这个树中了。

ImmediateEditor 的构造方法以一个树、一个绘制器和一个编辑器为参数，把这些参数传送给 DefaultTreeCellEditor 的构造方法。对这个绘制器的一个引用被保留，以便利用这个绘制器的 getCheckBoxOffset 方法。

```

class ImmediateEditor extends DefaultTreeCellEditor {
    private FileNodeRenderer renderer;

    public ImmediateEditor (JTree tree,
        FileNodeRenderer renderer,
        FileNodeEditor editor) {
        super (tree, renderer, editor);
        this.renderer = renderer;
    }
    ...
}

```

DefaultTreeCellEditor 实现一个 protected 方法，即 canEditImmediately 方法，它决定是否应该立即启动编辑。

ImmediateEditor 类重载 canEditImmediately 方法，如果传送给这个方法的事件是发生在这个绘制器的复选框区域内的鼠标单击事件的话，则返回 true。

```

...
protected boolean canEditImmediately (EventObject e) {
    boolean rv = false; // rv = return value

    if (e instanceof MouseEvent) {
        MouseEvent me = (MouseEvent) e;
        rv = inCheckBoxHitRegion (me);
    }

    return rv;
}

public boolean inCheckBoxHitRegion (MouseEvent e) {
    TreePath path = tree.getPathForLocation (e.getX (),
        e.getY ());
}

```

```

        FileNode node = (FileNode) path.getLastPathComponent ();
        boolean rv = false;
        if (node.isDirectory ()) {
            // offset and lastRow DefaultTreeCellEditor
            // protected members

            Rectangle bounds = tree.getRowBounds (lastRow);
            Dimension checkBoxOffset =
                renderer.getCheckBoxOffset ();

            bounds.translate (offset + checkBoxOffset.width,
                            checkBoxOffset.height);

            rv = bounds.contains (e.getPoint ());
        }
        return rv;
    }
}

```

树编辑器还负责确定事件是否应该选取节点。ImmediateEditor 重载 shouldSelectCell 方法，如果询问的这个节点是树叶，或在复选框外的文件夹节点中发生了鼠标单击的话，则返回 true。

```

public boolean shouldSelectCell (EventObject e) {
    boolean rv = false; // only mouse events
    if (e instanceof MouseEvent) {
        MouseEvent me = (MouseEvent) e;
        TreePath path = tree.getPathForLocation (me.getX (),
                                                    me.getY ());
        FileNode node = (FileNode)
            path.getLastPathComponent ();
        rv = node.isLeaf () || ! inCheckBoxHitRegion (me);
    }
    return rv;
}

```

组件总结 20-1 总结了 JTree 组件。

组件总结 20-1 JTree

模型:	javax.swing.tree.TreeModel
UI 代表:	javax.swing.plaf.basic.BasicTreeUI
绘制器:	TreeCellRenderer
编辑器:	CellEditor
激发的事件:	PropertyChangeEvents、TreeModelEvents、TreeSelectionEvents、 TreeExpansionEvents、ChangeEvents
替换:	——
类图:	见图 20-24

JTree 扩展 JComponent 并实现 Accessible 接口和 Scrollable 接口。除树单元绘制器和编辑器外，它还有两种模型：树模型和树选取模型。

JTree 类还维护一个哈希表和许多代表树属性的 integer 和 boolean 值。TreeSelectionRedirector 是一个 protected JTree 内部类，它响应由树的选取模型激发的选取事件，即把该事件传送给树的选取监听器。

cellEditor——缺省时，树是不可编辑的，因此它有一个 null 单元编辑器。如果用 JTree.setEditable 方法使树的编辑有效，则这个树将配备一个 DefaultTreeCellEditor 实例。

cellRenderer——缺省时，所有的树都配备一个 DefaultTreeCellRenderer 实例。DefaultTreeCellRenderer 类扩展 JLabel，因此，缺省时树单元绘制器是标签。

editable——树被选定后，必须通过调用 JTree.setEditable (true) 来显式地允许进行树编辑。

invokesStopCellEditing——通常，调用 cancelCellEditing () 或单击编辑的节点外的区域将可以取消对树单元格的编辑。缺省时，当取消编辑时，所作的任何修改都将丢失。然而，如果把 invokesStopCellEditing 属性设置为 true，则在取消编辑时将调用 stopCellEditing () 来保存所做的编辑修改。

largeModel——由树的界面样式使用，来确定是否应该优化树的显示。对含有大量数据的树而言，把 largeModel 属性设置为 true 可提高绘制性能和减少内存占用量。

界面样式不必适应树的 largeModel 属性，节点显示被优化的方式与界面样式有关。缺省时，如果树的 largeModel 属性为 true，则所有 Swing 提供的界面样式都使用固定高度的布局缓存。否则，则使用可变高度的布局缓存。注意，指定大模型只在树节点有固定高度时才起作用。

model——缺省时树模型是 DefaultTreeModel 的一个实例。

rootVisible——确定是否显示树的根节点。如果 rootVisible 属性为 false，则根节点的子节点沿着这个树的左边显示，请参见例 20-12 “显示根节点和根句柄”。

rowHeight——因为大多数树都有固定行高，所以 JTree 类允许显式地设置行高。如果 rowHeight 的值小于或等于 0，则树的单元绘制器为树的每一行计算高度。如果 rowHeight 被设置为大于 0 的值，则不计算行高。

scrollsOnExpand——缺省时，当展开文件夹时，树会被滚动以使尽可能多的子节点是可见的。将 ScrollsOnExpand 属性设置为 false 可取消这一功能。

selectionModel——树的选取模型实现 TreeSelectionModel 接口，缺省时，树的选取模型是 DefaultTreeSelectionModel 的一个实例。如果用 JTree.setSelectionModel (null) 将树的选取模型设置为 null，则树将配备一个共享的选取模型，这个模型不允许进行选取。

showsRootHandles——所有树文件夹都有一个控件来切换文件夹的展开状态。根节点是唯一能够通过调用 JTree.showsRootHandles () 来隐藏其句柄的文件夹。

toggleClickCount——缺省时，通过双击鼠标来实现文件夹的展开和折叠。toggleClickCount 属性允许显式地设置鼠标的点击数。

在 Swing1.1 FCS 中，还没有 toggleClickCount 属性，即它将在未来的版本中实现。

visibleRowCount——当把树放在滚动窗格中时的可见行数，缺省值为 20。

20.8.6 树事件

树模型激发事件以响应对节点的修改、节点的删除和插入。JTree 类激发事件以便响应事件选取和节点展开或折叠。当编辑停止或取消时，树单元编辑器将激发事件，表 20-3 列出了一些更有趣的树事件。

表 20-3 树事件

事件	由 ... 激发	当 ... 激发
TreeModelEvent	树模型	改变节点、插入节点、删除节点、改变树结构
TreeSelectionEvent	JTree	清除选取、改变选取路径
TreeExpansionEvent	JTree	展开文件夹、将展开文件夹、折叠文件夹、将折叠文件夹
ChangeEvent	DefaultCellEditor	停止或取消编辑

1. 树鼠标事件

除了表 20-3 列出的事件以外，树还激发鼠标事件。图 20-25 示出的小应用程序包含一个配备了鼠标监听器的树，这个监听器在树中发生鼠标按下事件时将更新这个小应用程序的状态区。

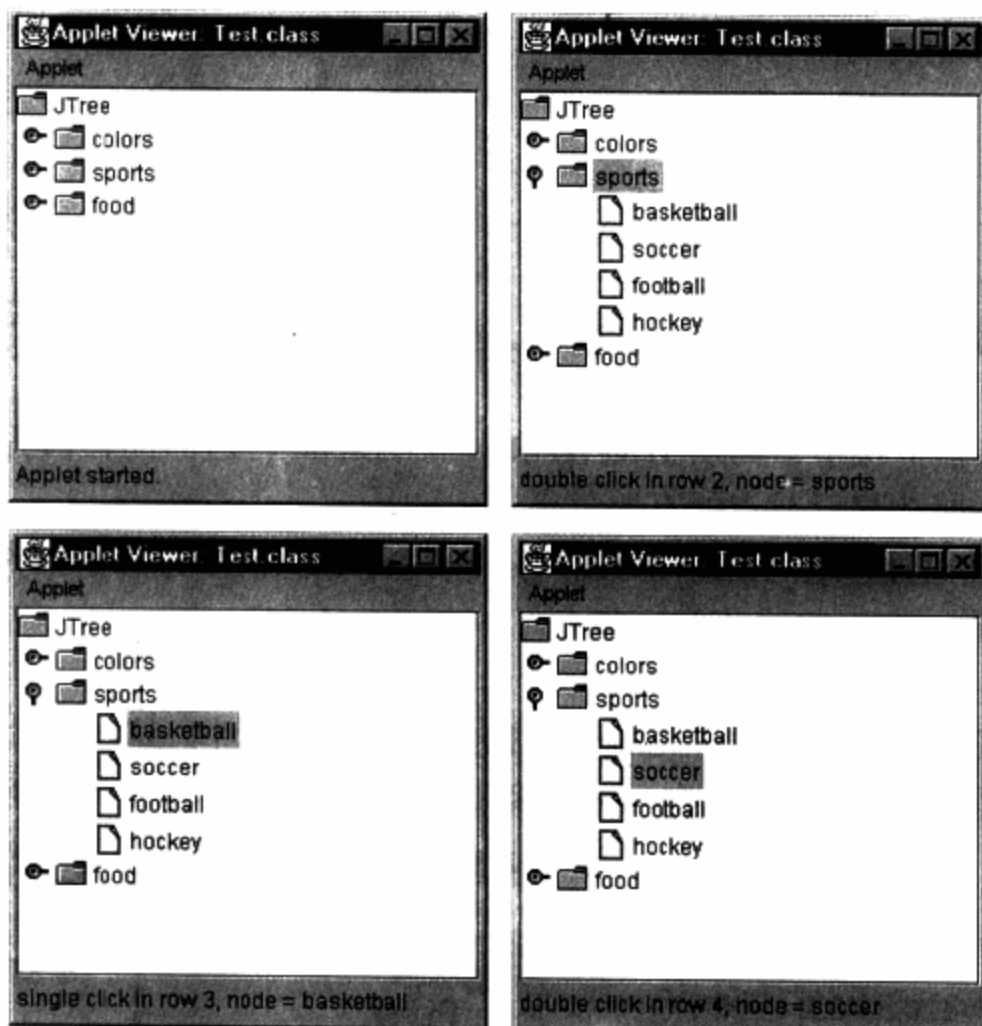


图 20-25 处理树鼠标事件

例 20-16 列出了图 20-25 所示的小应用程序的代码。

例 20-16 处理树鼠标事件

```
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public Test () {
        JTree tree = new JTree ();
        getContentPane ().add (new JScrollPane (tree));
        tree.addMouseListener (new MouseAdapter () {
            public void mousePressed (MouseEvent e) {
                String s = null;
                JTree t = (JTree) e.getSource ();
                int row = t.getRowForLocation (e.getX (), e.getY ());
                if (e.getClickCount () == 2)
                    s = "double click in row " + row;
                else
                    s = "single click in row " + row;
                if (row != -1) {
```



```

import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    public void init () {
        JTree tree = new JTree ();
        getContentPane().add (tree, BorderLayout.CENTER);

        // must invoke setEditable, or the call below to
        // getCellEditor () will return null.

        tree.setEditable (true);

        tree.getCellEditor ().addCellEditorListener (
            new CellEditorListener () {
                public void editingCanceled (ChangeEvent e) {
                    CellEditor editor = (CellEditor) e.getSource ();
                    String s = (String) editor.getCellEditorValue ();
                    showStatus ("editing cancelled: " + s);
                }

                public void editingStopped (ChangeEvent e) {
                    CellEditor editor = (CellEditor) e.getSource ();
                    String s = (String) editor.getCellEditorValue ();
                    showStatus ("editing stopped: " + s);
                }
            }
        );
    }
}

```

这个小应用程序实现了两个内部类监听器，它们从编辑器（事件源）获取编辑的值。`CellEditor.getCellEditorValue` 方法返回编辑节点的用户对象，在本例中是一个字符串。

3. 树选取事件

树选取事件由实现 `TreeSelectionListener` 接口的对象来处理，接口总结 20-8 总结了 `TreeSelectionListener` 接口。

接口总结 20-8 `TreeSelectionListener`

```
public abstract void valueChanged (TreeSelectionEvent e)
```

`TreeSelectionListener` 接口只定义了一个方法，它以 `TreeSelectionEvent` 的一个实例为参数。`TreeSelectionEvent` 类提供了大量有关选取事件的信息，但对树选取监听器而言，只有一个方法是必须的。

类总结 20-5 总结了 `TreeSelectionEvent` 类。

类总结 20-5 `TreeSelectionEvent`

扩展：`java.util.EventObject`

1. 构造方法

```

public TreeSelectionEvent (Object source, TreePath path, boolean isNew,
                           TreePath oldLeadSelectionPath,
                           TreePath newLeadSelectionPath)
public TreeSelectionEvent (Object source, TreePath [] paths, boolean [] areNew,

```

```
TreePath oldLeadSelectionPath,  
TreePath newLeadSelectionPath)
```

TreeSelectionEvent 提供了两个构造方法来构造只有一条路径或有一个路径数组的事件。TreeSelectionEvents 由树选取模型来创建，开发人员很少需要将 TreeSelectionEvents 实例化。

2. 方法

```
public TreePath getNewLeadSelectionPath ()  
public TreePath getOldLeadSelectionPath ()  
public TreePath getPath ()  
public TreePath [] getPaths ()  
public boolean isAddedPath ()  
public boolean isAddedPath (TreePath)  
public Object cloneWithSource (Object source)
```

上面所列的前两种方法返回对树路径的引用，这些树路径代表新的和旧的引导选取路径。引导选取路径被定义为添加到选取中的最后路径。

getPath 方法返回第一个选取路径，getPaths 方法返回一个选取路径数组。isAddedPath 方法确定最后的选取是否导致添加路径或从选取项中删除。isAddedPath (TreePath) 方法确定特定的选取路径是否添加或从选取中删除。

cloneWithSource 方法返回事件的拷贝，该方法把指定的对象作为这个事件的源。

图 20-27 所示的小应用程序包含一个分隔窗格，这个窗格包含一个树和一个文本区。当在这个树中做了选取时，这个选取事件的有关信息将添加到文本区中。

图 20-27 中的左上角的图片示出了在选取 blue 节点后小应用程序的样子。新的引导选取路径是从根节点到 blue 节点的路径，而旧的引导选取路径是 null，因为还没有前一次的选取。这个事件包含一条路径，即被添加到选取中的新引导选取路径。

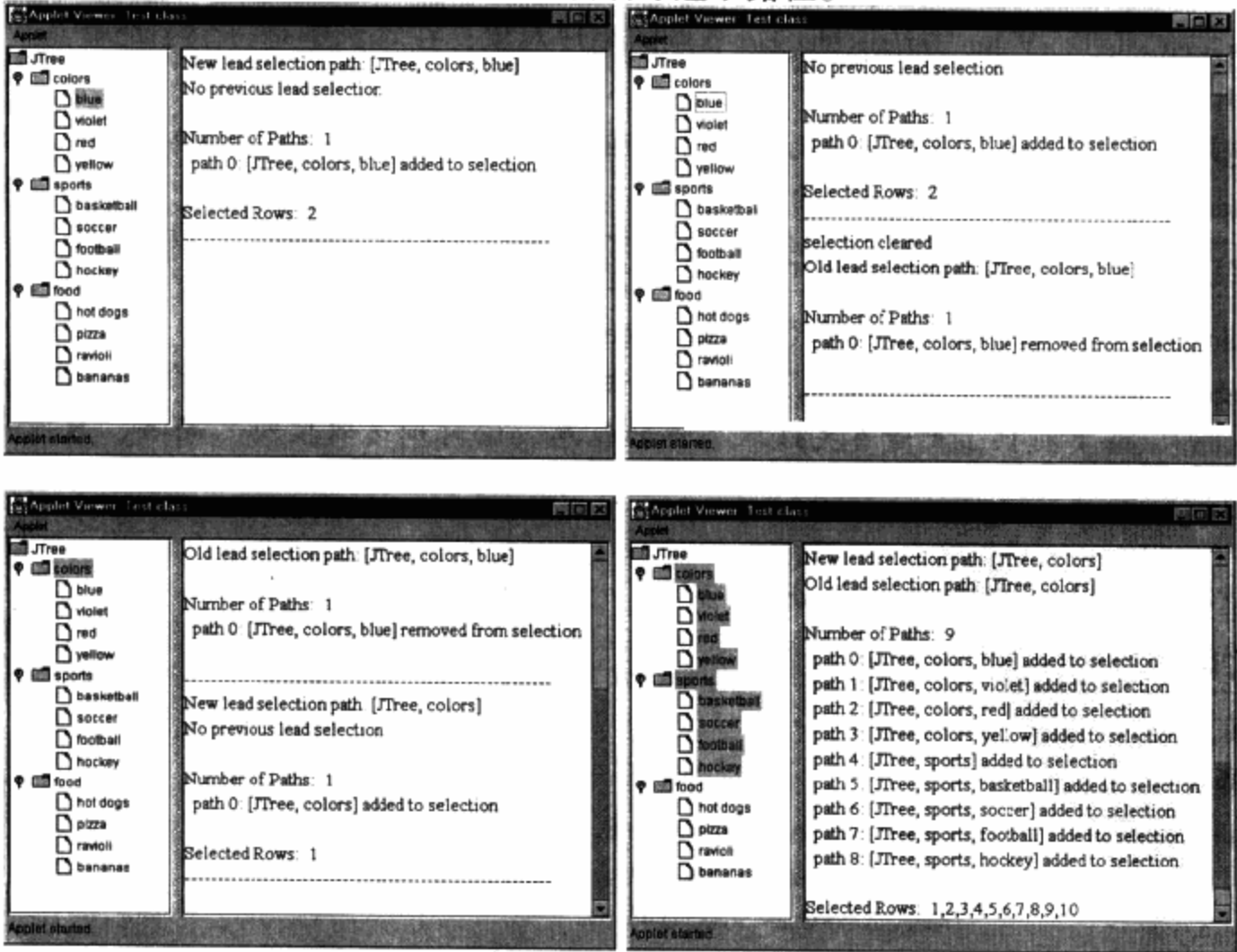


图 20-27 树选取事件

图 20-27 右上角的图片示出了取消选取 blue 节点（按下 Ctrl 键，然后单击 blue 节点）后这个小应用程序的样子。现在新的引导选取路径是 null，指示选取已经被清除。这时事件含有一条路径，即从选取中删除的旧引导选取路径。

图 20-27 左下角的图片示出了随后选取 colors 节点后小应用程序的样子。现在新的引导选取路径是从根节点到 colors 节点的一条路径，这个事件只包含一条代表新引导选取路径的路径。

图 20-27 右下角的图片示出了在选定 colors 节点到 hockey 节点之间的全部节点（按下 Shift 键，然后单击 hockey 节点）后这个小应用程序的样子。现在新的引导选取路径和旧的引导选取路径相同，这是由 Swing1.1 FCS 的一个错误造成的。新的引导选取路径应该被更新，以代表新选取的节点。选取的路径数量为 9，每条路径指示一个由选取事件选定的节点。

这个小应用程序为树添加了一个树选取监听器，以获取对新的和旧的引导选取路径的引用。如果新的引导选取路径是 null，则选取已经清除。有关新的和旧的引导选取路径的信息随后添加到这个小应用程序的文本区中，然后调用这个监听器的 printSelectionInformation () 方法，它显示有关树选取的附加信息。

```
public class Test extends JApplet {
    ...
    public void init () {
        ...
        tree.addTreeSelectionListener (
            new TreeSelectionListener () {
                public void valueChanged (TreeSelectionEvent e) {
                    TreePath path = e.getNewLeadSelectionPath ();
                    String s = new String ();
                    if (path != null) {
                        s += "New lead selection path: " +
                            path.toString () + "\n";
                    }
                    else
                        s += "selection cleared \n";
                    path = e.getOldLeadSelectionPath ();
                    if (path != null) {
                        s += "Old lead selection node:" +
                            path.toString () + "\n";
                    }
                    else
                        s += "No previous lead selection \n";
                    textArea.append (s + "\n");
                    printSelectionInformation (e);
                }
            }
        );
        ...
    }
}
```

printSelectionInformation 方法从事件中获取树的路径，并把有关每条路径的信息添加到文本区中。用 TreeSelectionEvent.isAddedPath 方法来确定是否把一个路径添加到选取中或从选取中删除一个路径。

```
...
void printSelectionInformation (TreeSelectionEvent e) {
    showPaths (e);
    showRows ();
}
```

```

        textArea.append (" \ n-----");
        textArea.append ("-----");
        textArea.append ("----- \ n");
    }
    private void showPaths (TreeSelectionEvent e) {
        TreePath [] paths = e.getPaths ();
        textArea.append ("Number of Paths: " +
            paths.length + " \ n");
        for (int i=0; i < paths.length; ++i) {
            TreePath path = paths [i];
            boolean wasAdded = e.isAddedPath (path);
            textArea.append (" path " + i + ": ");
            textArea.append (path +
                (wasAdded ? " added to selection" :
                " removed from selection") + " \ n");
        }
    }
    ...

```

这个监听器的 `showRows` 方法把有关这个树当前选定行的信息添加到文本区中。用 `JTree.getSelectionRows` 方法从这个树中获取代表选定行的一个 `integer` 数组。

```

    ...
    private void showRows () {
        int [] rows = tree.getSelectionRows ();
        if (rows != null && rows.length > 0) {
            textArea.append (" \ nSelected Rows: ");
            for (int i=0; i < rows.length; ++i) {
                textArea.append (
                    Integer.toString (rows [i]));
                if (i != rows.length-1)
                    textArea.append (",");
            }
        }
    }
}

```

例 20-18 列出了图 20-27 所示的小应用程序的代码。

例 20-18 处理树选取事件

```

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    JTree tree = new JTree ();
    JTextArea textArea = new JTextArea ();
    JSplitPane splitPane = new JSplitPane (
        JSplitPane.HORIZONTAL_SPLIT,

```



```

        new JScrollPane (tree),
        new JScrollPane (textArea));

public void init () {
    splitPane.setDividerLocation (150);

    getContentPane ().add (splitPane, BorderLayout.CENTER);

    tree.addTreeSelectionListener (
        new TreeSelectionListener () {
            public void valueChanged (TreeSelectionEvent e) {
                TreePath path = e.getNewLeadSelectionPath ();
                String s = new String ();

                if (path != null) {
                    s += "New lead selection path: " +
                        path.toString () + " \n";
                }
                else
                    s += "selection cleared \n";

                path = e.getOldLeadSelectionPath ();
                if (path != null) {
                    s += "Old lead selection node: " +
                        path.toString () + " \n";
                }
                else
                    s += "No previous lead selection \n";

                textArea.append (s + " \n");
                printSelectionInformation (e);
            }

            void printSelectionInformation (TreeSelectionEvent e) {
                showPaths (e);
                showRows ();

                textArea.append (" \n-----");
                textArea.append ("-----");
                textArea.append ("----- \n");
            }

            private void showPaths (TreeSelectionEvent e) {
                TreePath [] paths = e.getPaths ();

                textArea.append ("Number of Paths: " +
                    paths.length + " \n");

                for (int i=0; i < paths.length; ++i) {
                    TreePath path = paths [i];
                    boolean wasAdded = e.isAddedPath (path);

                    textArea.append (" path " + i + ": ");
                    textArea.append (path +
                        (wasAdded ? " added to selection" :
                        " removed from selection") + " \n");
                }
            }

            private void showRows () {
                int [] rows = tree.getSelectionRows ();

                if (rows != null && rows.length > 0) {
                    textArea.append (" \nSelected Rows: ");

```

```

        for (int i=0; i < rows.length; ++i) {
            textArea.append (
                Integer.toString (rows [i]));
            if (i != rows.length-1)
                textArea.append (",");
        }
    }
}

```

4. 树展开事件

文件夹被展开或被折叠后, JTree 类向所有已登记的监听器发送一个树展开事件, 这些监听器实现 TreeExpansionListener 接口。接口总结 20-9 总结了 TreeExpansionListener 接口。

接口总结 20-9 TreeExpansionListener

```

public abstract void treeCollapsed (TreeExpansionEvent)
public abstract void treeExpanded (TreeExpansionEvent)

```

TreeExpansionListener 接口定义了上述两个方法。读者可以练习一下, 看文件夹扩展或折叠之后哪个方法被调用。

由 TreeExpansionListener 接口定义的这两个方法都以一个 TreeExpansionEvent 实例为参数。类总结 20-6 总结了 TreeExpansionEvent 类。

类总结 20-6 TreeExpansionEvent

1. 构造方法

```

Public TreeExpansionEvent (Object source, TreePath
path)

```

用事件源来实例化 TreeExpansionEvent, 在发生扩展或折叠时这个事件总是 JTree 的一个实例。这个唯一的 TreeExpansionEvent 构造方法还带入了 TreePath 的一个实例, 这个实例代表扩展被折叠的节点。

2. 方法

```

public TreePath getPath ()

```

树展开事件中唯一可用的信息是到扩展或折叠文件夹的路径。

图 20-28 所示的小应用程序通过更新状态区来响应文件夹的展开和折叠。例 20-19 列出了图 20-28 所示小应用程序的代码。

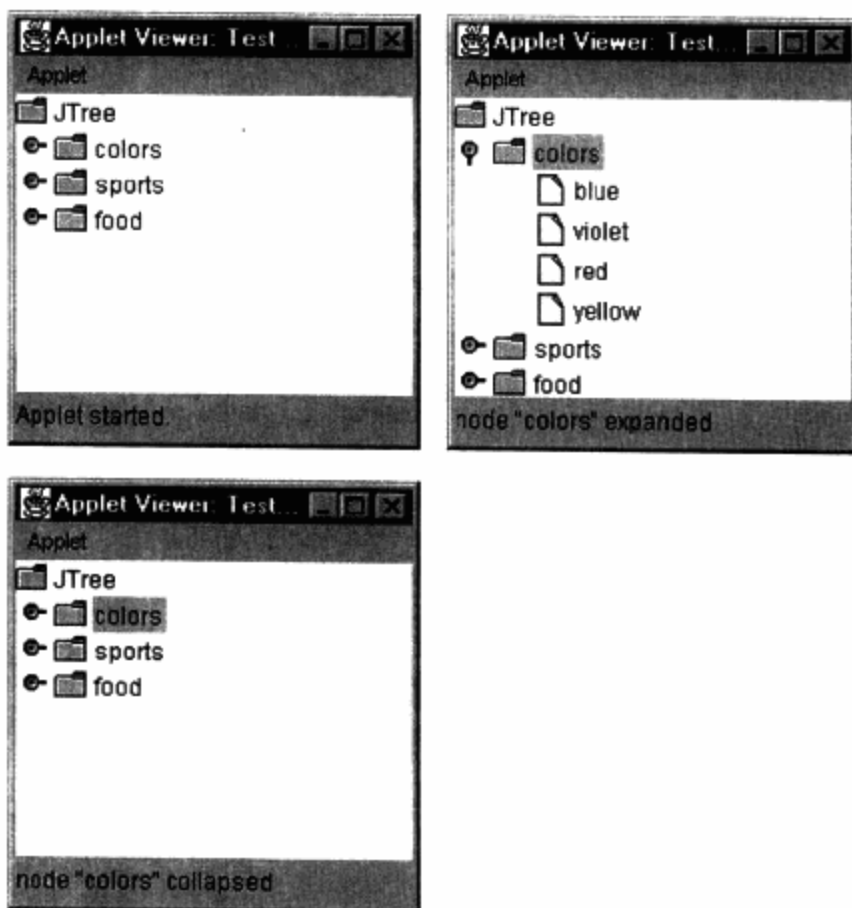


图 20-28 处理树展开事件

例 20-19 处理树展开事件

```

import javax.swing.*;

```

```

import javax.swing.event.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    public void init () {
        Container contentPane = getContentPane ();
        JTree tree = new JTree ();
        contentPane.add (tree);
        tree.addTreeExpansionListener (
            new TreeExpansionListener () {
                public void treeCollapsed (TreeExpansionEvent e) {
                    TreePath path = e.getPath ();
                    TreeNode node = (TreeNode)
                        path.getLastPathComponent ();
                    showStatus ("node " + "\ " + node.toString () +
                        "\ " + " collapsed");
                }
                public void treeExpanded (TreeExpansionEvent e) {
                    TreePath path = e.getPath ();
                    TreeNode node = (TreeNode)
                        path.getLastPathComponent ();
                    showStatus ("node " + "\ " + node.toString () +
                        "\ " + " expanded");
                }
            }
        );
    }
}

```

这个小应用程序把一个树展开监听器添加到树中，以获取展开或折叠的文件夹。监听器从事件中获取树的路径，然后通过路径获取节点，并用这个节点来更新状态区。

5. 否决节点展开和折叠

实际上，每次当展开或折叠树文件夹时，JTree 类都激发两个事件。在展开和折叠文件夹之前，要把一个树展开事件发送给已登记的监听器（这些监听器实现 TreeWillExpandListener 接口）。如果所有监听器都不否决这个展开或折叠，则执行展开或折叠，并把另一个树展开事件发送给已登记的监听器（这些监听器实现 TreeExpansionListener 接口）。

接口总结 20-10 总结了 TreeWillExpandListener 接口。

接口总结 20-10 TreeWillExpandListener

```

public abstract void treeWillCollapse (TreeExpansionEvent) throws ExpandVetoException;
public abstract void treeWillExpand (TreeExpansionEvent) throws ExpandVetoException;

```

TreeWillExpandListener 接口定义了上面所列的这两个方法，它们只在文件夹折叠或展开之前被调用。这两个方法都以一个 TreeExpansionEvent 为参数，并且这两个方法都可能弹出一个展开否决异常信息。

类总结 20-7 总结了 ExpandVetoException 类。

类总结 20-7 ExpandVetoException

构造方法

```
public ExpandVetoException (TreeExpansionEvent)
public ExpandVetoException (TreeExpansionEvent, String)
```

与大多数异常类一样，ExpandVetoException 也提供了构造方法，但没有提供方法。用一个 TreeExpansionEvent 和一个代表消息的字符串（可选）来构造 ExpandVetoException。应该注意的是，消息（如果有的话）不能被 Swing 类所使用。

图 20-9 所示的小应用程序包含一个配备了 TreeWillExpandListener 的树，这个 TreeWillExpandListener 否决展开 colors 文件夹，并否决折叠 food 文件夹。

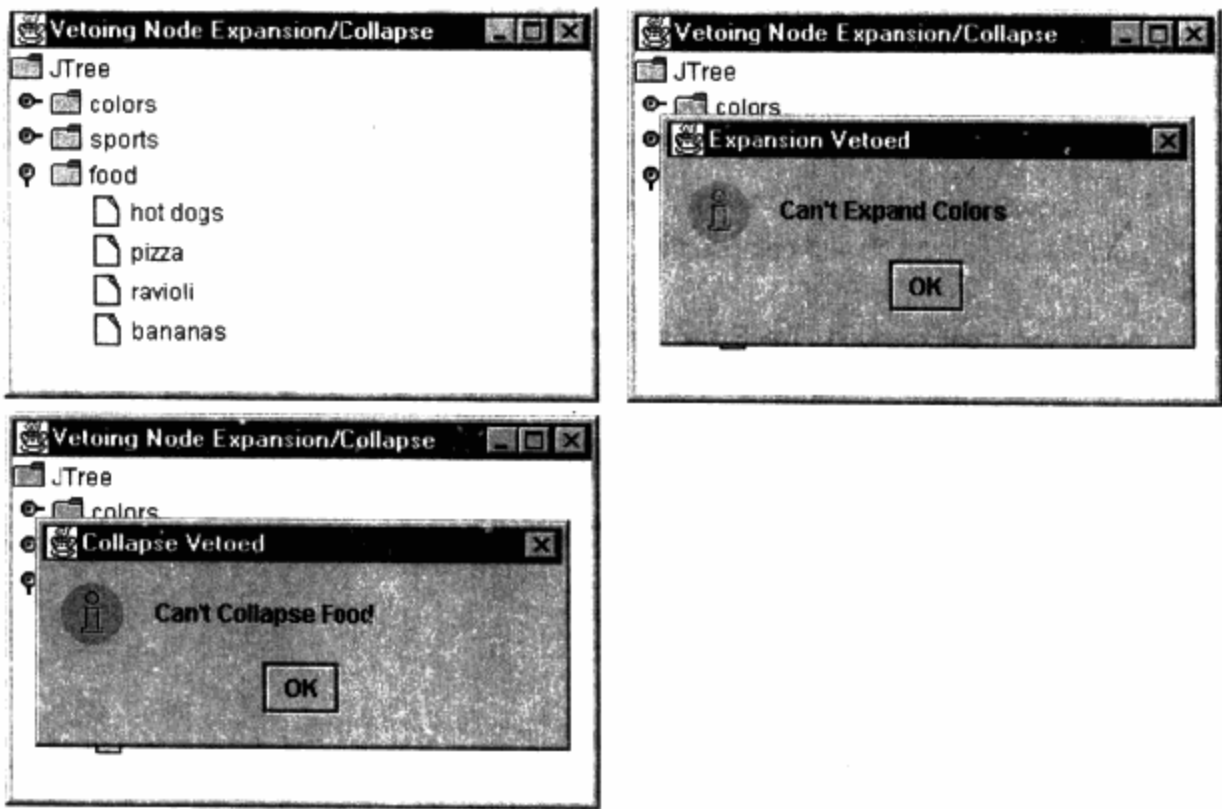


图 20-29 否决节点的展开和折叠

例 20-20 列出了图 20-29 所示的小应用程序的代码。

例 20-20 否决节点的展开和折叠

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JFrame {
    public Test () {
        JTree tree = new JTree ();
        getContentPane ().add (tree);

        tree.addTreeWillExpandListener (
            new TreeWillExpandListener () {
                public void treeWillExpand (TreeExpansionEvent e)
                    throws ExpandVetoException {
                    TreePath path = e.getPath ();
                    TreeNode node = (TreeNode)
                        path.getLastPathComponent ();
                    if (node.toString ().equals ("colors")) {
```

```

        JOptionPane.showMessageDialog (Test.this,
            "Can't Expand Colors",
            "Expansion Vetoed",
            JOptionPane.INFORMATION_MESSAGE);
        throw new ExpandVetoException (e);
    }

    public void treeWillCollapse (TreeExpansionEvent e)
        throws ExpandVetoException {
        TreePath path = e.getPath ();
        TreeNode node = (TreeNode)
            path.getLastPathComponent ();
        if (node.toString ().equals ("food")) {
            JOptionPane.showMessageDialog (Test.this,
                "Can't Collapse Food",
                "Collapse Vetoed",
                JOptionPane.INFORMATION_MESSAGE);
            throw new ExpandVetoException (e);
        }
    }

    };

    public static void main (String args []) {
        GraphicJavaApplication.launch (new Test (),
            "Vetoing Node Expansion/Collapse", 300, 300, 300, 200);
    }
}

```

这个监听器的 `treeWillExpand` 方法和 `treeWillCollapse` 方法分别获取将展开和将折叠的节点，并通过弹出一个展开否决错误信息，来否决展开或折叠。

20.8.7 JTree 类总结

类总结 20-8 列出了 JTree 的 public 和 protected 变量和方法。

类总结 20-8 JTree

1. 常量

```

public static final String CELL_EDITOR_PROPERTY
public static final String CELL_RENDERER_PROPERTY
public static final String EDITABLE_PROPERTY
public static final String INVOKES_STOP_CELL_EDITING_PROPERTY
public static final String LARGE_MODEL_PROPERTY
public static final String ROOT_VISIBLE_PROPERTY
public static final String ROW_HEIGHT_PROPERTY
public static final String SCROLLS_ON_EXPAND_PROPERTY
public static final String SELECTION_MODEL_PROPERTY
public static final String SHOWS_ROOT_HANDLES_PROPERTY
public static final String TREE_MODEL_PROPERTY
public static final String VISIBLE_ROW_COUNT_PROPERTY

```

上面所列的这些常量代表 JTree 的约束属性。在属性变化监听器的 `propertyChange` 方法中可

用它们来确定哪个属性被修改了。

2. 构造方法

```
public JTree ()
public JTree (Object [])
public JTree (Hashtable)
public JTree (Vector)
public JTree (TreeModel)
public JTree (TreeNode)
public JTree (TreeNode, boolean asksAllowsChildren)
```

JTree 类提供了带数据或不带数据来创建树的构造方法。用无参数的构造方法创建的树配备了一组缺省节点。这种构造方法主要用于 JavaBean 构造器，在其他地方很少使用。

有些构造方法以对象数组、一个哈希表或一个矢量为参数来创建树，但在实际中很少使用这些构造方法。大多数树都是用上面所列的最后三个构造方法中的一个方法来创建的。

3. 方法

(1) Scrollable 接口方法

```
public Dimension getPreferredSize ()
public int getScrollableBlockIncrement (Rectangle, int orientation, int direction)
public boolean getScrollableTracksViewportHeight ()
public boolean getScrollableTracksViewportWidth ()
public int getScrollableUnitIncrement (Rectangle, int orientation, int direction)
```

JTree 类实现 Scrollable 接口，有关 Scrollable 接口的详细内容，请参见 13.3 节“Scrollable 接口”。

当把一个树放在滚动窗格中时，用 getPreferredSize 方法来为它的视口定义树的首选大小。首选视口高度是通过树的可视行数和行高来计算的。

当树滚动了一个块增量时，getScrollableBlockIncrement 方法返回滚动树所需的像素数，块增量等于树的可视矩形的宽度或高度（取决于滚动方向参数）。

当树滚动了一个单元增量时，getScrollableUnitIncrement 方法返回滚动树所需的像素数，单元增量的高度是把下一个节点滚入视图所需的像素数，单元增量的宽度为 4 个像素。

getScrollableTracksViewportHeight 和 getScrollableTracksViewportWidth 方法分别确定树的高度和宽度是否与树的视口一致。如果树比视口小，则返回 true，否则返回 false。这两个方法用于确保树的大小永远不比它的视口小。

(2) 树模型

```
protected static TreeModel getDefaultTreeModel ()
protected static TreeModel createTreeModel (Object)
protected TreeModelListener createTreeModelListener ()
public void setModel (TreeModel)
public TreeModel getModel ()
public void setLargeModel (boolean)
public boolean isLargeModel ()
```

当用无参数的 JTree 构造方法构造树时，就使用 getDefaultTreeModel 方法来创建缺省的模型。当用带有 Object 数组、哈希表或矢量等参数的 JTree 构造方法构造树时，就使用 createTreeModel 方法来创建树模型。

createTreeModelListener 方法创建 JTree.TreeModelHandler 的一个实例，它通过更新树中节点

的展开状态来响应插入节点或删除节点。

JTree 类还为其模型提供了 `setModel` 和 `getModel` 方法，它们分别用于设置和获取模型。

如果树含有大量数据，则可调用 `setLargeModel (true)` 以便加快树的显示。这种加快是由树的界面样式来实现的，并不是所有的界面样式都支持一个大模型。

(3) 树选取

```
public void addTreeSelectionListener (TreeSelectionListener)
public void removeTreeSelectionListener (TreeSelectionListener)
public void addSelectionInterval (int, int)
public void addSelectionPath (TreePath)
public void addSelectionPaths (TreePaths [])
public void addSelectionRow (int)
public void addSelectionRows (int [])
public void clearSelection ()

public Object getLastSelectedPathComponent ()
public TreePath getLeadSelectionPath ()
public int getLeadSelectionRow ()
public int getMaxSelectionRow ()
public int getMinSelectionRow ()
public int getSelectionCount ()
public TreePath getSelectionPath ()
public TreePaths [] getSelectionPaths ()
public int [] getSelectionRows ()

public boolean isSelectionEmpty ()

public void removeSelectionInterval (int, int)
public void removeSelectionPath (TreePath)
public void removeSelectionPaths (TreePath [])
public void removeSelectionRow (int)
public void removeSelectionRows (int [])

public void setSelectionInterval (int, int)
public void setSelectionMode ()
public void setSelectionMode (TreeSelectionMode)
public void setSelectionPath (TreePath)
public void setSelectionPaths (TreePath [])
public void setSelectionRow (int)
public void setSelectionRows (int [])
protected void fireValueChanged (TreeSelectionEvent)
```

JTree 类提供了大量与树选取相关的方法。上面所列的许多方法直接授权给树的选取模型，可以设置选取间隔、路径和行，也可以把选取间隔、路径和行添加到当前的选取中，或从当前的选取中删除选取间隔、路径和行。`clearSelection` 方法可以清除选取。

(4) 树展开

```
public void addTreeExpansionListener (TreeExpansionListener)
public void addTreeWillExpandListener (TreeWillExpandListener)

public void removeTreeExpansionListener (TreeExpansionListener)
public void removeTreeWillExpandListener (TreeWillExpandListener)

public void fireTreeCollapsed (TreePath)
public void fireTreeExpanded (TreePath)
public void fireTreeWillCollapse (TreePath) throws ExpandVetoException;
```



```

public void fireTreeWillExpand (TreePath) throws ExpandVetoException;
public void collapsePath (TreePath)
public void collapseRow (int row)
public void expandPath (TreePath)
public void expandRow (int row)

public boolean hasBeenExpanded (TreePath)
public boolean isCollapsed (int row)
public boolean isCollapsed (TreePath)
public boolean isExpanded (int row)
public boolean isExpanded (TreePath)
public void clearToggledPaths ()
public void setExpandedState (TreePath, boolean)

public boolean getScrollsOnExpand ()
public void setScrollsOnExpand (boolean)

```

JTree 类提供了添加和删除 TreeExpansionListeners 和 TreeWillExpandListeners 的方法，有关处理树展开事件的更多信息，请参见 20.8.6 节。还提供了把树展开事件发送给监听器的一些方法。因为 TreeWillExpandListeners 可以否决文件夹展开和折叠，所以 fireTreeWillCollapse 和 fireTreeWillExpand 方法可能弹出异常信息。

通过程序也可以展开或折叠树的文件夹，JTree 类还提供了许多用于确定是否展开或折叠行或路径的方法。

缺省时，当树被放在滚动窗格中，并且文件夹被展开时，树会被滚动以尽可能多地显示这个文件夹的子节点，调用 setScrollsOnExpand (false) 可以关闭这一功能。

(5) 树单元编辑和绘制

```

public TreeCellEditor getCellEditor ()
public TreeCellRenderer getCellRenderer ()
public void setCellEditor (TreeCellEditor)
public void setCellRenderer (TreeCellRenderer)

public boolean isEditable ()
public void setEditable (boolean)

public boolean getInvokesStopCellEditing ()
public void setInvokesStopCellEditing (boolean)

public void startEditingAtPath (TreePath)
public boolean isEditing ()

public boolean stopEditing ()
public void cancelEditing ()

```

树有一个单独的绘制器和一个单独的编辑器，并且 JTree 类为它们提供了访问方法。缺省时树是不可编辑的，因此，要编辑一个 JTree 实例必须调用 setEditable (true) 方法。

缺省时，当激活一个树编辑器而另一个节点被选取时，编辑就会取消。此时可以调用 setInvokesStopCellEditing (true) 来强制保存所做的修改。

startEditingAtPath 方法启动给定节点的编辑。只有当树的编辑器确定该节点与可编辑路径一致时才启动编辑。开发人员很少调用 startEditingAtPath 方法。

isEditing 方法确定树当前是否正在编辑节点。StopEditing 和 cancelEditing 方法分别用于停止和取消编辑。

(6) 树路径

```

protected TreePath [] getPathBetweenRows (int index0, int index1)

```

```

public Rectangle getPathBounds (TreePath)
public TreePath getClosestPathForLocation (int x, int y)
public TreePath getPathForLocation (int, x, int y)
public TreePath getPathForRow (int)
protected Enumeration getDescendantToggledPaths (TreePath)
public TreePath getEditingPath ()
public Enumeration getExpandedDescendants (TreePath)
public boolean isPathEditable (TreePath)
public boolean isPathSelected (TreePath)
public boolean isVisible (TreePath)

public void makeVisible (TreePath)
public void scrollPathToVisible (TreePath)
protected void removeDescendantToggledPaths (Enumeration)

```

上述方法用于获取行的路径和位置，并确定路径的特性，如路径是可视还是被选取的。例如，`getPathForLocation` 方法有一个位置参数，如果这个位置与节点一致，则这个方法返回 `true`，否则返回 `null`。`getClosestPathForLocation` 返回距离给定位置最近的路径，而不考虑是否与节点一致。

`makeVisible` 方法展开由传送给它的路径所标识的文件夹，并确保节点是可视的；如果把树放在滚动窗格中，则 `scrollPathToVisible` 把节点滚动到视图。

(7) 树的行

```

public Rectangle getRowBounds (int row)
public int getRowCount ()
public int getRowForLocation (int x, int y)
public int getClosestRowForLocation (int x, int y)
public int getRowForPath (TreePath)

public boolean isFixedRowHeight ()
public boolean isRowSelected (int row)

public int getRowHeight ()
public void setRowHeight (int row)

public int getVisibleRowCount ()
public void setVisibleRowCount (int row)

public void scrollRowToVisible (int row)

```

`JTree` 类提供了许多处理行的方法。例如，除了可获得与给定位置相对应的行以外，还可以获取行的范围。当把树放在一个滚动窗格中时，可视的行数是可以设置的，而且可以把某行滚动到视图中。

(8) 实用方法

```

public String convertValueToText (JTree tree, Object value, boolean isSelected,
                                   boolean expanded, boolean leaf, int row)
public boolean getShowsRootHandles ()
public void setShowsRootHandles (boolean)
public void setRootVisible (boolean)
public boolean isRootVisible ()
public String getToolTipText (MouseEvent)
protected String paramString ()
public void treeDidChange ()

```

`JTree` 类提供了许多实用方法。对绘制器而言，`convertValueToText` 方法把传送给绘制器的值

转换为文本。为根节点的可视性和根节点句柄提供了访问方法。有关根节点的可见性和根节点句柄的详细内容，请参见 20.6.3 节“根节点和根节点句柄”。

重载 JComponent 的 `getToolTipText` 方法，这个方法允许显示绘制器的工具提示。有关 `getToolTipText` 方法的更多信息，请参见 4.7 节“工具提示”。

重载 Object 类的 `paramString` 方法，这个方法返回树的文本描述。为树的 UI 代表提供了 `treeDidChange` 方法，开发人员不能直接调用这个方法。

(9) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()  
public TreeUI getUI ()  
public String getUIClassID ()  
public void setUI (TreeUI)  
public void updateUI ()
```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。`updateUI` 方法在组件配备了 UI 代表时调用。

20.8.8 AWT 兼容

AWT 不提供与 JTree 类似的组件。

20.9 本章回顾

与 JTable 一样，JTree 是 Swing 最复杂的组件之一。JTree 是一个精心设计的组件，尽管它很复杂，但却很容易使用。当展开或折叠节点和滚动时，JTree 也提供了极好的性能。

第 21 章 文本基础

Swing 文本组件是相对比较简单组件。它建立在由 javax.swing.text 包的类和接口提供的一个复杂的下层构件之上。本书将分三章来介绍 Swing 文本。本章主要讨论所有文本组件从 JTextComponent 类继承的基本功能。在第 22 章的“文本组件”中将讨论这些文本组件。在第 23 章的“定制文本组件”中将讨论 Swing 文本的更高级的主题，如视图、元素、属性集和风格。

21.1 Swing 文本组件

Swing 提供了五种文本组件，它们都列在表 21-1 中。所有的 Swing 文本组件最终都是 JComponent 类的扩展，JComponent 类为它们提供了许多基本功能。JComponent 类的总结参见 21.7 的“JTextComponent”。

表 21-1 Swing 文本组件

组件	多行文本	简单/多风格的文本	使用 ...
JTextArea	●	简单	用于处理短的可编辑文档，该文档只有一种字体，可以选取字环绕
JTextField		简单	用于输入单行文本
JPasswordField		简单	用于输入一行密码
JEditorPane	●	简单/多风格	用于浏览具有不同格式的文本
JTextPane	●	多风格	用于处理具有多种字体、嵌入图片或组件的文档

虽然所有的 Swing 文本组件都是 JTextComponent 类的扩展并且使用 Swing.Text 包中的方法，但是 Swing 文本组件可以分为两种类型：简单文本组件和多风格文本组件。

在表 21-1 中列出的前 3 类组件（JTextArea、JTextField、JPasswordField）都是简单文本组件，它们一次只能显示一种字体和一种颜色。JTextArea 和 JTextField 与 AWT 的 TextArea 和 TextField 组件保持兼容。

在表 21-1 中列出的后两类组件 JEditorPane 和 JTextPane 都是多风格的组件，它们一次能显示多种字体和颜色。JEditorPane 组件能够显示不同格式的文本，如 HTML 和 RTF；JTextPane 组件能够嵌入图片和组件。

图 21-1 是一个类图，它提供对 Swing 文本组件的一个概览。Swing 文本组件没有直接实现滚动，而是由 JTextComponent 类来实现 Scrollable 接口，以便可以把文本组件放到滚动窗格中。

JTextArea、JTextField、JEditorPane 直接扩展 JComponent 类，其中 JTextField 的扩展可以输入密码。

与 JTextField 及其扩展不同的是，JTextArea 类能够显示多行文本，当到达字或字符的边界时，能够自动换行。JEditorPane 类保持了一张哈希表，它将不同的文本格式与它的编辑程序关联起来，从而可以显示不同格式的文本。JTextPane 是 JEditorPane 的一个扩展。



图 21-1 Swing 文本组件的概览

Swing 文本包

Swing 文本包提供了许多类和接口，这些类和接口支持文本组件。表 21-2 列出了 Swing 文本包定义的主要类和接口。

表 21-2 Swing 文本的基本类和接口

名字	类和接口	描述
AttributeSet	接口	元素的键或值的属性集
EditorKit	类	提供对文本组件进行处理的动作集。为多风格的文本组件创建视图和文档
View	类	绘制部分或全部文本组件的内容
Document	类	保存文本组件的内容和内容的属性
Element	接口	代表文档中具有属性的一段内容
Keymap	接口	击键与动作的对应集
Caret	接口	文本的脱字符
Highlighter	接口	在文本组件中绘制增亮区
Position	接口	表示文档中的位置
Style	接口	具有名字、可变的属性集
StyleContext	类 ^①	在文本组件中共享的属性
ViewFactory	接口	为元素创建视图库

① StyleContext 实现了 AbstractDocument、AttributeContext 的接口

在表 21-2 列出的类和接口将在这一章和下两章详细讨论。下面的类和接口可以通过 `JTextComponent` 类访问，它们将在本章介绍。

- Document
- Keymap
- Caret
- Highlighter
- Position

21.2 动作

文本组件的许多功能都封装在组件编辑软件工具包所提供的动作集中。使用 `JTextComponent.getActions` 方法可以获取文本组件的动作，该方法返回一个动作数组。

将文本组件的功能封装在动作中主要有两点好处。首先，可以在多个组件中共享动作。实际上，所有的文本组件都共享缺省的动作集。我们可以在 5.3 节“动作”中了解这方面的详细信息。其次，我们可以很容易地将击键、工具栏按钮、菜单项和动作关联在一起。

21.2.1 文本动作

在缺省的情况下，文本组件的动作是 `TextActions` 类的一个实例。`TextActions` 类扩展了 `AbstractAction`。回顾 5.3.1 节的“作为控制中心点的动作”，我们可以知道动作不只是和一个文档相关联，动作可以被多个文档共享。因此，当执行一个文档动作时，也就是说当 `actionPerformed` 方法被调用时，这个动作必须找到执行动作的文本对象。

`TextAction` 类提供一个 `protected getTextComponent` 方法，动作事件传递给该方法。如果是文本组件激发事件，`getTextComponent()` 方法将返回该组件。否则，方法将返回最后有焦点的文本组件。

下面列出的 `DefaultEditorKit` 类的一段代码，该代码描述了 `TextAction` 的扩展怎样使用 `TextAction.getTextComponent` 方法。

```
//From DefaultEditorKit...

public static class CutAction extends TextAction {
    public CutAction () {
        super (cutAction);
    }
    public void actionPerformed (ActionEvent e) {
        JTextComponent target = getTextComponent (e);
        if (target != null) {
            target.cut ();
        }
    }
}
```

`CutAction` 类是 `TextAction` 扩展中的一个。它是由 `DefaultEditorKit` 类实现的。在表 21-3 可以找到这些类。传递给父类 `TextAction` 的构造方法的 `CutAction` 变量是 `DefaultEditorKit` 中定义的字符串，它指出动作是剪切动作。

类总结 21-1 `TextAction`

扩展: `AbstractAction`

1. 构造方法

```
public TextAction (String name)
```

TextAction 类提供一个构造方法，其输入参数为动作的名字。动作的名字将往上传递给 AbstractAction 的构造方法

2. 方法

```
public static final Action [] augmentList (Action [], Action [])
protected final JTextComponent getFocusedComponent ()
protected final JTextComponent getTextComponent (ActionEvent)
```

TextAction 类提供了一个方便的静态方法，往一个动作数组中加入另一个动作数组。使用 augmentList 方法，可以很方便地往文本组件的缺省动作中加入动作。在例 23-4 中可以看到怎样使用 TextAction.augmentList 方法。

getFocusedComponent 方法返回最后有焦点的文本组件。最后有焦点的文本组件实际上是由 JTextComponent 类维护的。在 21.7 节“JTextComponent”中讨论了 JTextComponent 的方法。

如果激发事件的是文本组件的话，getTextComponent 方法将返回激发事件的文本组件。否则，将返回最后有焦点的文本组件。

在图 21-2 中显示的小应用程序，包含一个分割窗格，它的左边是文本域，右边是文本域的动作列表。在动作列表中选取一个动作将使该动作在文本域上执行。图 21-2 的左图显示了在文本域上选取了一部分文字时小应用程序的状态，右图则显示了当执行剪切到剪切板的动作时小应用程序的状态。

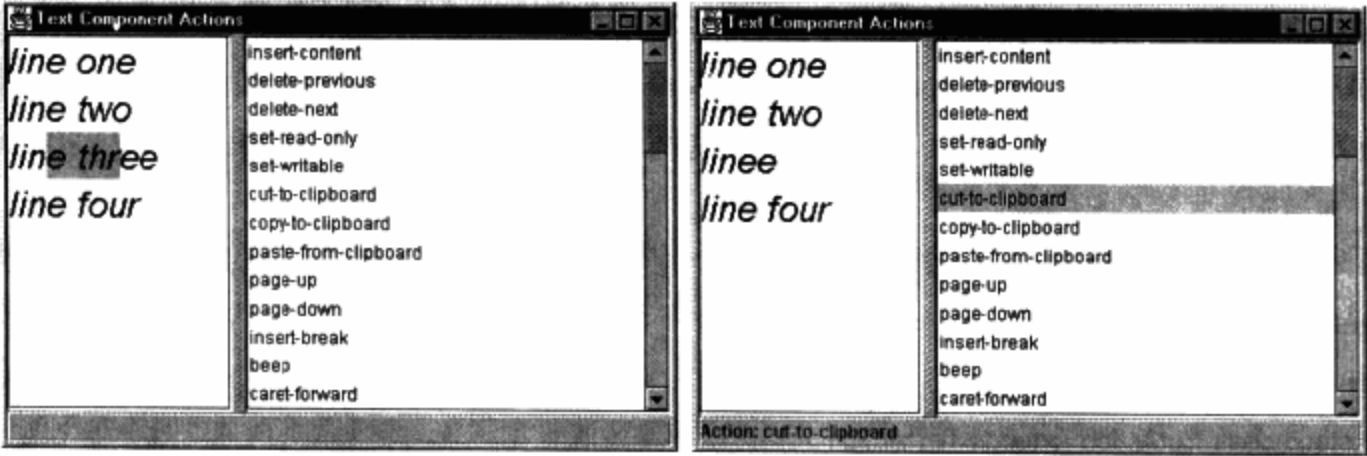


图 21-2 文本组件的动作

应用程序创建了文本域和列表，把它们添加到一个分割的窗格。然后，分割窗格作为一个中央组件加到应用程序的内容窗格。小应用程序安装了 GJApp 状态区，将其作为内容窗格的下部组件。在 2.2 节“GJApp”可以找到关于 GJApp 的更多信息。当执行某个动作时，小应用程序将用动作的名字更新状态区。

文本域的动作通过调用 getActions () 方法得到。把动作传递到小应用程序的 createActionList 方法，该方法把每一个动作加到小应用程序的列表中。

```
public class Test extends JFrame {
    private JTextArea textArea = createTextArea ();
    private Action [] actions = textArea.getActions ();

    private JList actionList = createActionList (actions);
    private JSplitPane splitPane = new JSplitPane (
        JSplitPane.HORIZONTAL_SPLIT,
        new JScrollPane (textArea),
        new JScrollPane (actionList));
```



```

public Test () {
    Container contentPane = getContentPane ();
    ...
    contentPane.add (splitPane, BorderLayout.CENTER);
    contentPane.add (GJApp.getStatusArea (),
        BorderLayout.SOUTH);
}
private JList createActionList (Action [] actions) {
    DefaultListModel model = new DefaultListModel ();
    final JList list = new JList (model);

    ...
    for (int i=0; i < actions.length; ++i) {
        model.addElement (actions [i]);
    }
    ...
    list.addListSelectionListener (new ListSelectionListener () {
        public void valueChanged (ListSelectionEvent e) {
            if (! e.getValueIsAdjusting ()) {
                Action source =
                    (Action) actionList.getSelectedValue ();
                source.actionPerformed (null);
                GJApp.showStatus ("Action:" +
                    (String) source.getValue (Action.NAME));
            }
        }
    });
    return list;
}
}

```

把动作加到应用程序列表的列表选取监听器所激发。监听器获得所选取的动作的引用，并以一个 null 事件调用 actionPerformed 方法，使最后有焦点的文本组件执行该动作。

例 12-1 列出了图 21-2 小应用程序的代码。

例 21-1 使用文本组件动作

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.util.*;

public class Test extends JFrame {
    private JTextArea textArea = createTextArea ();
    private Action [] actions = textArea.getActions ();

    private JList actionList = createActionList (actions);
    private JSplitPane splitPane = new JSplitPane (
        JSplitPane.HORIZONTAL_SPLIT,
        new JScrollPane (textArea),
        new JScrollPane (actionList));

    public Test () {
        Container contentPane = getContentPane ();
    }
}

```

```

splitPane.setDividerLocation (150);

contentPane.add (splitPane, BorderLayout.CENTER);
contentPane.add (GJApp.getStatusArea (),
                 BorderLayout.SOUTH);

|
private JList createActionList (Action [] actions) {
    DefaultListModel model = new DefaultListModel ();
    final JList list = new JList (model);

    list.setSelectionMode (
        ListSelectionMode.SINGLE_SELECTION);

    for (int i = 0; i < actions.length; ++i) {
        model.addElement (actions [i]);
    }

    list.setCellRenderer (new DefaultListCellRenderer () {
        public Component getListCellRendererComponent (
            JList list, Object value,
            int index, boolean isSelected,
            boolean cellHasFocus) {
            super.getListCellRendererComponent (list, value,
                index, isSelected, cellHasFocus);

            Action action = (Action) value;
            setText ( (String) action.getValue (Action.NAME));

            return this;
        }
    });

    list.addListSelectionListener (new ListSelectionListener () {
        public void valueChanged (ListSelectionEvent e) {
            if (! e.getValueIsAdjusting ()) {
                Action source =
                    (Action) actionList.getSelectedValue ();

                textArea.requestFocus ();
                source.actionPerformed (null);

                GJApp.showStatus ("Action: " +
                    (String) source.getValue (Action.NAME));
            }
        }
    });

    return list;
}

private JTextArea createTextArea () {
    JTextArea textArea = new JTextArea (
        "line one \nline two \nline three \nline four");

    textArea.setFont (new Font ("Dialog", Font.ITALIC, 24));
    textArea.getCaret ().setBlinkRate (0);
    return textArea;
}

public static void main (String args []) {
    GJApp.launch (new Test (),
        "Text Component Actions", 300, 300, 450, 300);
}

```

21.2.2 动作和编辑工具包

正如在 21.2 节“动作”中所提到的，文本组件的动作由组件的编辑工具包来维护。DefaultEditorKit 类提供了一个为所有文本组件所共享的缺省动作集，每一个动作都有相应的名字。

图 21-3 显示的小应用程序说明了如何使用编辑工具包来访问与文本组件相关联的特定动作。这个小应用程序创建了一个由动作和它们名字构成的哈希表，并使用在 DefaultEditorKit 类定义的字符串常量来访问剪切、拷贝和粘贴动作。

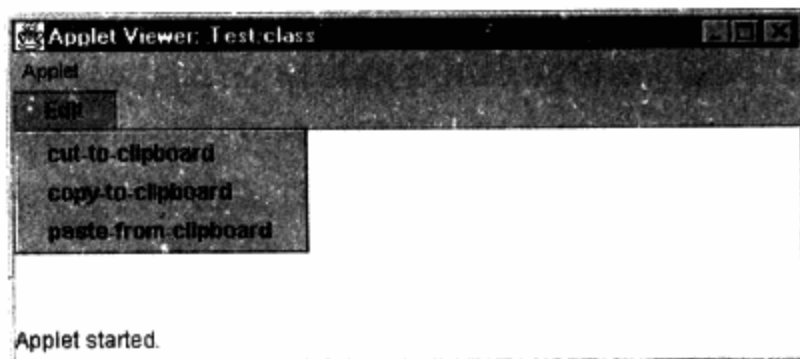


图 21-3 文本组件的动作

例 21-2 列出了图 21-3 所示小应用程序的代码。

例 21-2 用缺省编辑工具包访问动作

```
import java.awt.* ;
import javax.swing.* ;
import javax.swing.text.* ;
import java.util.* ;

public class Test extends JApplet {
    private JTextArea textArea = new JTextArea ("some content");
    private Hashtable actionTable = new Hashtable ();

    public void init () {
        Container contentPane = getContentPane ();
        textArea.setFont (new Font ("Dialog", Font.PLAIN, 24));
        loadActionTable ();
        setJMenuBar (createMenu ());
        contentPane.add (textArea, BorderLayout.CENTER);
    }

    private void loadActionTable () {
        Action [] actions = textArea.getActions ();
        for (int i=0; i < actions.length; ++i) {
            actionTable.put (actions [i] .getValue (Action.NAME),
                             actions [i]);
        }
    }

    private Action getAction (String name) {
        return (Action) actionTable.get (name);
    }

    private JMenuBar createMenu () {
        JMenuBar menuBar = new JMenuBar ();
        JMenu editMenu = new JMenu ("Edit");

        editMenu.add (getAction (DefaultEditorKit.cutAction) );
        editMenu.add (getAction (DefaultEditorKit.copyAction) );
        editMenu.add (getAction (DefaultEditorKit.pasteAction) );
        menuBar.add (editMenu);
    }
}
```

```
return menuBar;
```

这个小应用程序包含一个文本域，而且维护了一个由文本域的动作及动作名字构成的哈希表。这个小应用程序还实现了一个 `getAction` 方法，这个方法以动作名字为参数并返回与名字相关联的相应动作。

通过调用这个小应用程序的 `getAction` 方法来把动作添加到 `Edit` 菜单项中，在 `DefaultEditorKit` 中定义的剪切、拷贝、粘贴动作的常量字符串将传递给 `getAction` 方法。

`DefaultEditorKit` 类还提供许多用于某些动作的类。例 21-2 列出的小应用程序能够创建新动作，它被添加到 `Edit` 菜单中，如下所示：

```
editMenu.add (new DefaultEditorKit.CutAction ());
editMenu.add (new DefaultEditorKit.CopyAction ());
editMenu.add (new DefaultEditorKit.PasteAction ());
```

表 21-3 列出了由 `DefaultEditorKit` 定义的字符串常量和类。

表 21-3 `DefaultEditorKit` 的动作

类 ^①	字符串	类 ^①	字符串
	<code>backwardAction</code>		<code>pasteAction</code>
<code>BeepAction</code>	<code>beepAction</code>		<code>previousWordAction</code>
	<code>beginAction</code>		<code>readOnlyAction</code>
	<code>beginLineAction</code>		<code>selectAllAction</code>
	<code>beginParagraphAction</code>		<code>selectLineAction</code>
	<code>beginWordAction</code>		<code>selectParagraphAction</code>
<code>CopyAction</code>	<code>copyAction</code>		<code>selectWordAction</code>
<code>CutAction</code>	<code>cutAction</code>		<code>selectionBackwardAction</code>
<code>DefaultKeyTypedAction</code>	<code>defaultKeyTypedAction</code>		<code>selectionBeginAction</code>
	<code>deleteNextCharAction</code>		<code>selectionBeginLineAction</code>
	<code>deletePrevCharAction</code>		<code>selectionBeginParagraphAction</code>
	<code>downAction</code>		<code>selectionBeginWordAction</code>
	<code>endAction</code>		<code>selectionDownAction</code>
	<code>endLineAction</code>		<code>selectionEndAction</code>
	<code>endParagraphAction</code>		<code>selectionEndLineAction</code>
	<code>endWordAction</code>		<code>selectionEndParagraphAction</code>
	<code>forwardAction</code>		<code>selectionEndWordAction</code>
<code>InsertBreakAction</code>	<code>insertBreakAction</code>		<code>selectionForwardAction</code>
<code>InsertContentAction</code>	<code>insertContentAction</code>		<code>selectionNextWordAction</code>
<code>InsertTabAction</code>	<code>insertTabAction</code>		<code>selectionPreviousWordAction</code>
	<code>nextWordAction</code>		<code>selectionUpAction</code>
	<code>pageDownAction</code>		<code>upAction</code>
	<code>pageUpAction</code>		<code>writableAction</code>

① 所有的类是由 `DefaultEditorKit` 嵌套的类。

Swing 提示**文本动作和编辑器工具包**

Swing 的文本组件的一个重要特性是许多文本组件的功能都被封装于一个动作集中。由于动作可以被多个文本组件所共享，而且最后有焦点的文本组件执行动作，所以动作集能够被不同类型的文本组件所共享。例如，表 21-3 列出的由 **DefaultEditorKit** 提供的动作集被所有的 Swing 文本组件共享。

21.3 键映射

也许文本组件最基本的任务是将击键与特定的动作相关联。每一个文本组件都维护一个键映射来得到击键与特定动作的对应关系。这个对应关系是在 **Keymap** 接口中定义的。我们可以通过 **JTextComponent** 的 **getKeymap()** 和 **setKeymap()** 方法来访问一个文本组件的键映射。

除了使用 **setKeymap()** 方法来设置文本组件的键映射外，还可以通过指定一个键映射及其父映射来把一个键映射添加到文本组件中。当某一个击键在子键映射中找不到它的定义时，将搜索父键映射来寻找它的定义。**JTextComponent** 类提供了许多方法来操纵文本组件的键映射。

接口总结 21-1 Keymap**1. 动作**

```
public abstract void addActionForKeyStroke (KeyStroke, Action)
public abstract Action getAction (KeyStroke)
public abstract Action [] getBoundAction ()
public abstract Action getDefaultAction ()
public abstract void setDefaultAction (Action)
```

addActionForKeyStroke 方法将击键/动作对添加到键映射，**getAction** 返回与某一击键相对应的动作。**getBoundAction** 方法返回包含在键映射中的一个动作数组，这个动作数组与某个击键相关联。每一个文本组件都有一个缺省的动作，它通常将击键添加到组件，缺省的动作通过上面列出的最后两个方法来访问。

2. 击键

```
public abstract KeyStroke [] getBoundKeyStrokes ()
public abstract KeyStroke [] getKeyStrokesForAction (Action)
public abstract boolean isLocallyDefined (KeyStroke)
public abstract void removeKeyStrokeBinding (KeyStroke)
```

在上面列出的方法都是用来处理键映射中的击键。**GetBoundKeyStrokes** 返回与某一动作相关联的击键数组。一个动作可以有多个击键与之对应，**getKeyStrokeForAction** 方法返回与某一动作相对应的击键数组。

IsLocallyDefined 方法返回一个布尔值，它说明了一个击键是在键映射中被定义还是在父键映射中被定义。**RemoveKeyStrokeBinding** 方法删除键映射中的一个击键。

3. 解析父/名字

```
public abstract String getName ()
public abstract Keymap getResolveParent ()
public abstract void setResolveParent (Keymap)
public abstract void removeBindings ()
```

每一个键映射都有一个名字和一个父映射，我们将父映射称为解析父。上面列出的前三个

方法是 `name` 和 `resolveParent` 属性的访问方法。`removeBindings` 通过删除所有的击键/动作捆绑来有效地删除一个键映射。

图 21-4 显示的小应用程序包含一个文本域和一个设置文本域键映射的复选框。当选中复选框时，文本域的键映射被设置，否则，删除文本域的键映射。

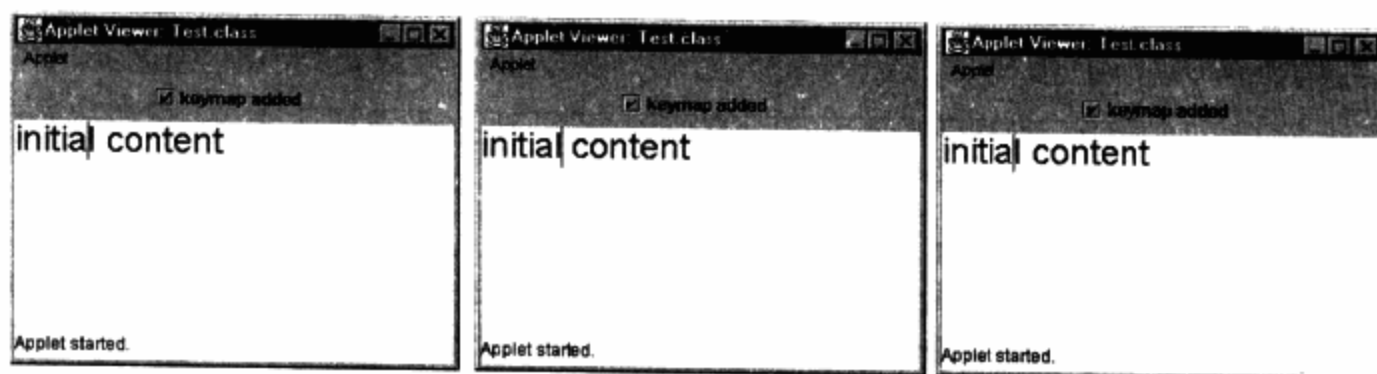


图 21-4 添加和删除键映射

当选中复选框时，文本域的键映射被设置。该键映射定义两对击键/动作捆绑（一个是按下 `Alt + F` 时，将加字符向前移动，另一个是按下 `Alt + B` 时，将加字符向后移动）。用这个小应用程序的 `createKeymap` 方法来建立键映射。

```
...
private Keymap createKeymap () {
    Keymap map = JTextComponent.addKeymap ("applet keymap",
                                           textArea.getKeymap ());

    KeyStroke forwardKeyStroke =
        KeyStroke.getKeyStroke (KeyEvent.VK_F,
                               InputEvent.ALT_MASK),
    backwardKeyStroke =
        KeyStroke.getKeyStroke (KeyEvent.VK_B,
                               InputEvent.ALT_MASK);

    Action forwardAction =
        getAction (DefaultEditorKit.forwardAction),
    backwardAction =
        getAction (DefaultEditorKit.backwardAction);

    map.addActionForKeyStroke (forwardKeyStroke,
                               forwardAction);
    map.addActionForKeyStroke (backwardKeyStroke,
                               backwardAction);

    return map;
}
...
```

键映射由 `JTextComponent.addKeymap` 方法建立。该方法的参数为键映射的名字和父映射。文本域的初始键映射作为父映射。

当创建键映射后，可以用 `Keymap.addActionForKeyStroke` 方法将击键与动作加入到键映射中。

与这个小应用程序的复选框相关联的动作监听器设置文本域的键映射。监听器根据复选框是否选中来设置文本域的键映射。

```
...
cbox.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        textArea.setKeymap (cbox.isSelected () ?
```

```

        newKeymap : originalKeymap);
    textArea.requestFocus ();
}
});
...

```

例 21-3 列出了图 21-4 所示小应用程序的完整代码。

例 21-3 设置键映射

```

import javax.swing. * ;
import javax.swing.text. * ;
import java.awt. * ;
import java.awt.event. * ;
import java.util. * ;

public class Test extends JApplet {
    private JTextArea textArea = new JTextArea ("initial content");
    private JCheckBox cbox = new JCheckBox ("keymap added");
    private Hashtable actionTable = new Hashtable ();
    private Keymap originalKeymap, newKeymap;

    public void init () {
        Container contentPane = getContentPane ();

        loadActionTable ();
        originalKeymap = textArea.getKeymap ();
        newKeymap = createKeymap ();

        textArea.setFont (new Font ("Dialog", Font.PLAIN, 24));
        contentPane.add (new ControlPanel (), BorderLayout.NORTH);
        contentPane.add (textArea, BorderLayout.CENTER);
    }

    private Keymap createKeymap () {
        Keymap map = JTextComponent.addKeymap ("applet keymap",
                                                textArea.getKeymap ());

        KeyStroke forwardKeyStroke =
            KeyStroke.getKeyStroke (KeyEvent.VK_F,
                                    InputEvent.ALT_MASK),
        backwardKeyStroke =
            KeyStroke.getKeyStroke (KeyEvent.VK_B,
                                    InputEvent.ALT_MASK);

        Action forwardAction =
            getAction (DefaultEditorKit.forwardAction),
        backwardAction =
            getAction (DefaultEditorKit.backwardAction);

        map.addActionForKeyStroke (forwardKeyStroke,
                                    forwardAction);
        map.addActionForKeyStroke (backwardKeyStroke,
                                    backwardAction);

        return map;
    }

    private void loadActionTable () {
        Action [] actions = textArea.getActions ();

        for (int i=0; i < actions.length; ++i) {
            actionTable.put (actions [i].getValue (Action.NAME),

```



```
        actions [i]);  
    }  
    }  
    private Action getAction (String name) {  
        return (Action) actionTable.get (name);  
    }  
    }  
    class ControlPanel extends JPanel {  
        public ControlPanel () {  
            add (cbox);  
            cbox.addActionListener (new ActionListener () {  
                public void actionPerformed (ActionEvent e) {  
                    textArea.setKeymap (cbox.isSelected () ?  
                        newKeymap : originalKeymap);  
                    textArea.requestFocus ();  
                }  
            });  
        }  
    }  
}
```

21.4 文档

与其他 Swing 组件一样，利用修改的模型-视图-控制器体系结构来实现文本组件。文本组件模型是 Document 接口的实现。文档维护文本组件的内容和与这个内容相关联的所有属性，这些属性如粗体、斜体等。当内容被改变时，文档还激发文档事件和撤销编辑事件。

图 21-5 中示出了文档的类层次。

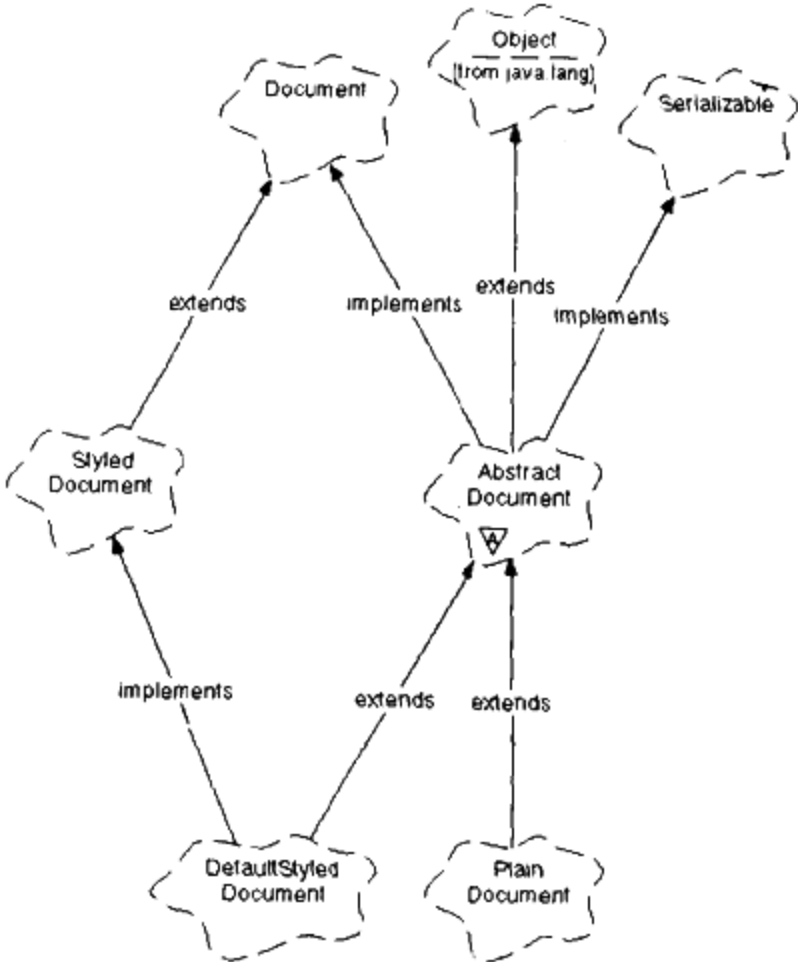


图 21-5 Swing 文档的类和接口

Document 接口由 AbstractDocument 类实现并由 StylableDocument 接口扩展。PlainDocument 类是

AbstractDocument 的一个扩展，与 DefaultStyledDocument 类一样，PlainDocument 类也实现 StyledDocument 接口。

文档的结构由元素的层次来维护，元素实现 Element 接口。每个元素都有一个与之相关联的属性集。元素和它的属性集将在第 23 章详细讨论。

文档还维护位置。位置置于文档内容的字符间，当改变文档时，它们的位置将被保持。图 21-6 显示的应用程序包含一个配备有定制视图的文本组件。这个视图图形地显示具有矩形位置定位的元素。

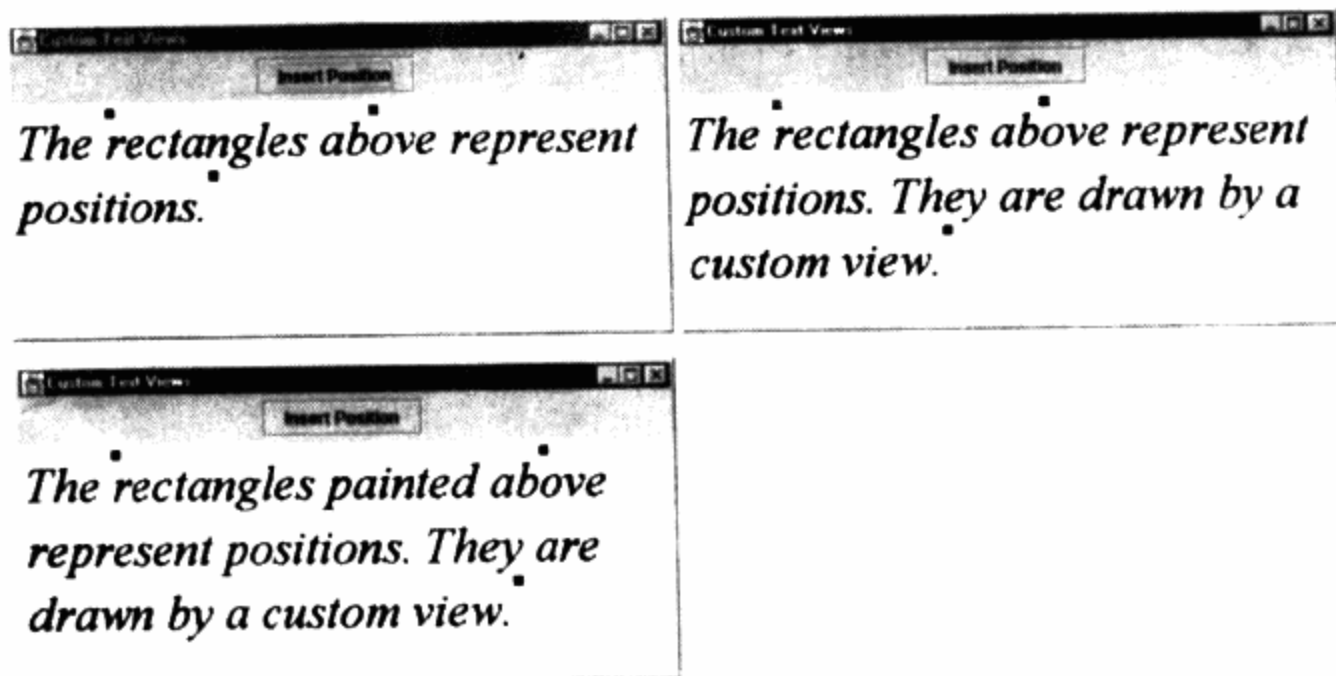


图 21-6 位置

图 21-6 显示的应用程序的代码没在这一章中列出，在 23.4 节“视图”中，介绍了这个应用程序并列出了它的代码。

接口总结 21-2 总结了 Document 接口。

接口总结 21-2 Document

1. 常量

```
public static final String StreamDescriptionProperty
public static final String TitleProperty
```

上面列出的常量字符串标识从中读取文档的流以及文档的名字。JTextComponent.read() 设置 StreamDescriptionProperty 常量。然而在 Swing 1.1 FCS 中，TitleProperty 不在 Swing 中使用。

在 Document.getProperty 和 Document.putProperty 方法中，常量被使用。

2. 方法

(1) 内容

```
public abstract String getText (int offset, int length) throws BadLocationException
public abstract void getText (int offset, int length, Segment) throws BadLocationException
public abstract void insertString (int offset, String, AttributeSet) throws BadLocationException
public abstract void remove (int offset, int length) throws BadLocationException
```

使用上面列出的前两个方法可以访问在文档中保存的文本。这两个方法都以相对于文档开始的起始位置 and 要获取的文本长度为参数。第一个方法返回一个字符串，第二个方法以一个 Segment 类的实例为参数。段提供了对一个字符数组的访问，不需要拷贝文本。

insertString 方法在指定的起始位置插入一个具有指定属性的字符串，如果起始位置无效，则弹出一个 BadLocationException 异常信息。

remove 方法从指定的起始位置移去文本的一段指定长度的内容。

(2) 位置

```
public abstract Position createPosition (int offset) throws BadLocationException
public abstract Position getStartPosition ()
public abstract Position getEndPosition ()
```

上面列出的第一个方法在文档中指定的起始位置创建一个位置。如果起始位置无效，则弹出一个 BadLocationException 异常信息。

使用上面列出的后两个方法可以访问文档的起始和结束位置。

(3) 根元素

```
public abstract Element getDefaultRootElement ()
public abstract Element [] getRootElements ()
```

使用上面列出的方法可以访问文档的缺省根元素和文档所维护的根元素数组。在 23.6 节“元素”中可以找到更多的关于文档和元素的信息。

(4) 文档和可撤销编辑监听器

```
public abstract void addDocumentListener (DocumentListener)
public abstract void addUndoableEditListener (UndoableEditListener)
public abstract void removeDocumentListener (DocumentListener)
public abstract void removeUndoableEditListener (UndoableEditListener)
```

使用上面列出的方法可以将文档和可撤销编辑监听器加入到文档和从文档中移走。在 21.4.2 节“文档监听器”中可以找到更多的关于文档监听器的信息。在 21.6 节“撤销/恢复”中可以找到更多的关于文本组件撤销/恢复实现的信息。

(5) 属性/绘制

```
public abstract Object getProperty (Object key)
public abstract void putProperty (Object key, Object value)
public abstract void render (Runnable)
```

所有的文档都维护一个属性集，它可以使用上面列出的前两个方法访问。render 方法的输入参数为 Runnable，执行 Runnable 从而使它可以在并行的情况下安全地读出文档。render 方法的输入参数 Runnable 不应修改文档。

21.4.1 定制文档

通常扩展文档从而提供定制行为，如有的文本域限制所能包含的字符数，有的文本域只允许特定类型的数据。

图 21-7 所示的小应用程序包含一个配备有定制文档的文本域，这个文本域只允许往文本域输入整数。

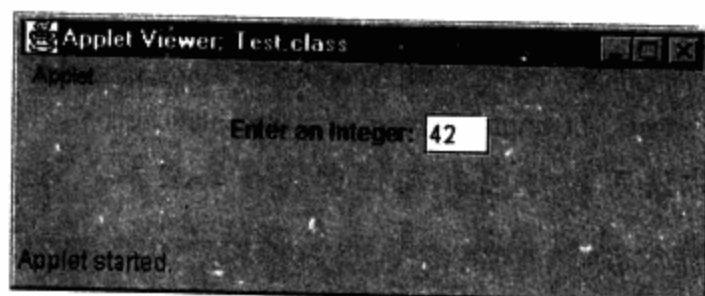


图 21-7 一个简单的定制文档

例 21-4 列出了图 21-7 所示小应用程序的代码。

例 21-4 只可以输入整数的文档

```

import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    JTextField tf = new JTextField (3);

    public Test () {
        Container contentPane = getContentPane ();
        JLabel label = new JLabel ("Enter an Integer:");
        tf.setDocument (new IntegerDocument ());
        contentPane.setLayout (new FlowLayout ());
        contentPane.add (label);
        contentPane.add (tf);
    }
}

class IntegerDocument extends PlainDocument {
    public void insertString (int offset, String s,
                             AttributeSet attributeSet)
        throws BadLocationException {
        try {
            Integer.parseInt (s);
        }
        catch (Exception ex) { // only allow integer values
            Toolkit.getDefaultToolkit ().beep ();
            return;
        }
        super.insertString (offset, s, attributeSet);
    }
}

```

这个小应用程序创建一个文本域，同时将文本域的文档设置为 `IntegerDocument` 的一个实例。

`IntegerDocument` 类扩展文本域的缺省文档 `PlainDocument`，并重载 `insertString` 方法。`IntegerDocument.insertString` 方法分析要插入的字符串。如果 `Integer.parseInt` 方法弹出一个异常信息，这个方法将使机器发声，然后不插入字符串就返回。如果字符串是一个有效的整数，该方法将调用 `super.insertString ()`。

21.4.2 文档监听器

用文档监听器可以跟踪文档的修改。接口总结 21-3 总结了 `DocumentListener` 接口，监听器接口在 `javax.swing.event` 包中。

接口总结 21-3 `DocumentListener`

```

public abstract void changedUpdate (DocumentEvent)
public abstract void insertUpdate (DocumentEvent)
public abstract void removeUpdate (DocumentEvent)

```

当文档属性被修改时（如往文档中插入内容或删除文档内容），将通知文档监听器。

在文档监听器接口中定义的每个方法的输入参数都是一个文档事件。接口总结 21-4 总结

了 DocumentEvent 接口。

接口总结 21-4 DocumentEvent

```
public abstract DocumentEvent.ElementChange getChange (Element)
public abstract Document getDocument ()
public abstract int getLength ()
public abstract int getOffset ()
public abstract DocumentEvent.EventType getType ()
```

上面列出的第一个方法返回 DocumentEvent.ElementChange 的一个实例。它说明了对文档元素作了什么改变。当调用文档监听器的 changeUpdate 方法时，使用 getChange 方法。

接下来的三个方法返回与事件相关联的文档以及所修改内容的起始位置和长度。

上面列出的最后一个方法返回事件的类型，它是 DocumentEvent.EventType 的一个实例。类总结 21-2 总结了 DocumentEvent.EventType。

类的总结 21-2 DocumentEvent.EventType

扩展： java.lang.Object

1. 常量

```
public abstract final DocumentEvent.EventType CHANGE
public abstract final DocumentEvent.EventType INSERT
public abstract final DocumentEvent.EventType REMOVE
```

DocumentEvent.EventType 类定义了三个常量，它们代表文档事件的类型。

2. 方法

```
public String toString ()
```

DocumentEvent.EventType 类也实现 toString 方法，该方法将返回代表事件类型的字符串。

图 21-8 中显示的小应用程序包含两个文本域。上面的文本域的文档包含应用程序的文本，下面的文本域显示对上面文本域文档所做的修改。应用程序也包含一个菜单，可以对文本进行剪切、拷贝和粘贴。编辑菜单有一个保存菜单项，当第一次修改文档后，将允许保存该文档。

左上图显示所选取的文本正被剪切到剪贴板，导致下面的文本域正被更新，如右上角的图片所示。注意，保存菜单项开始时是无效的，当对上面的文本域作了第一次修改后，就可以使用保存菜单项了。左下图显示把剪切到剪贴板的文本粘贴到文本域，导致更新下面的文本域，如右下图片所示。

这个应用程序的构造方法将文档监听器添加到上面的文本域中，它通过使保存菜单项有效和更新下面的文本域，来对文档的修改作出相应的反应。

```
public Test () {
    ...
    textArea.getDocument ().addDocumentListener (
        new DocumentListener () {
            public void insertUpdate (DocumentEvent e) {
                saveAction.setEnabled (true);
                updateStatus (e);
            }
            public void removeUpdate (DocumentEvent e) {
```

```
        saveAction.setEnabled (true);
        updateStatus (e);
    }

    public void changedUpdate (DocumentEvent e) {
        saveAction.setEnabled (true);
        updateStatus (e);
    }

    private void updateStatus (DocumentEvent e) {
        status.append (e.getType () .toString ());
        status.append (" Offset: " + e.getOffset ());
        status.append (" Length: " + e.getLength () + " \n");
    }
}

);
```

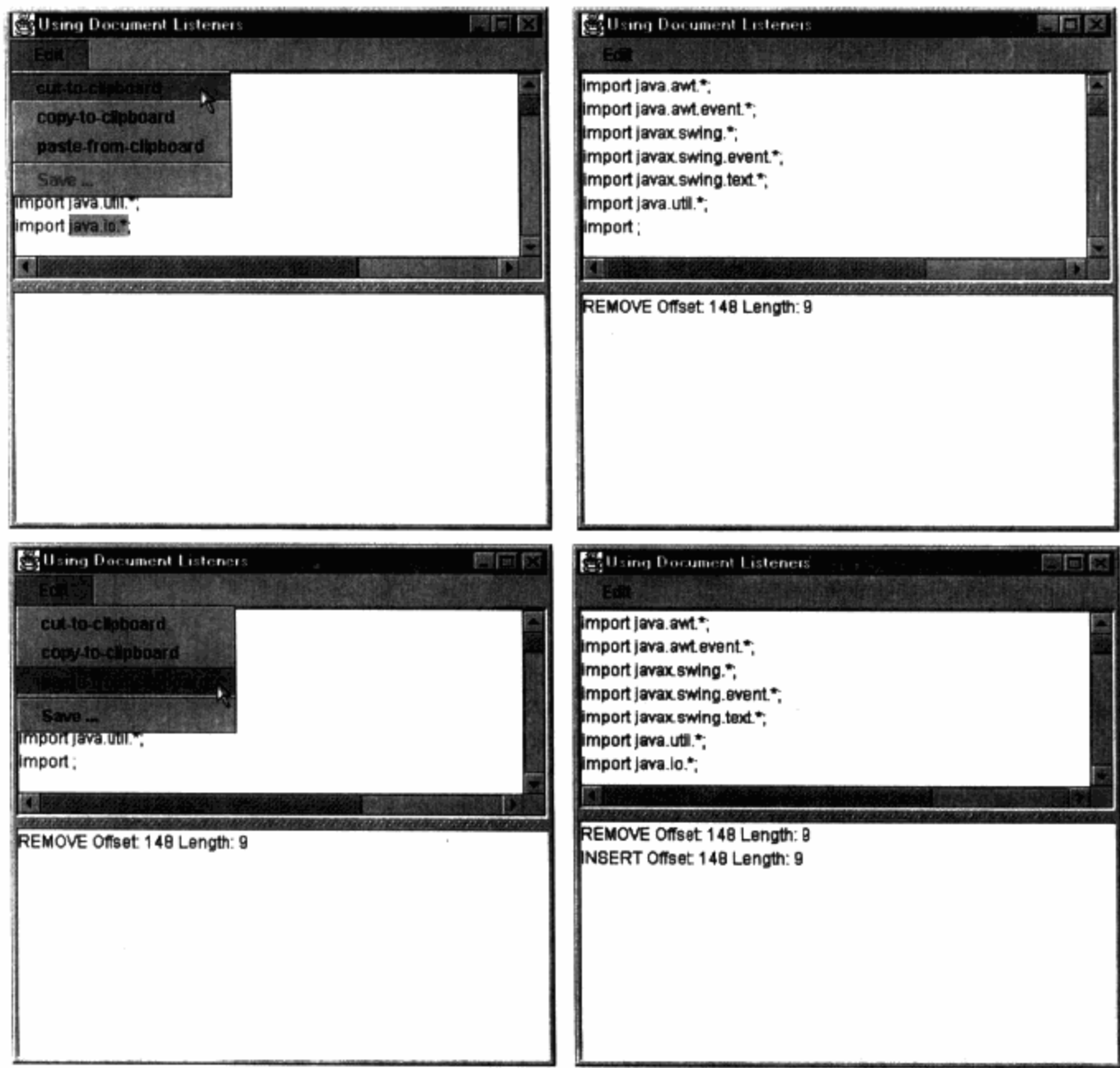


图 21-8 使用文档监听器

例 21-5 列出了图 21-8 所示的小应用程序的完整代码。

例 21-5 使用文档监听器

```
import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.event. * ;
import javax.swing.text. * ;
import java.util. * ;
```

```

import java.io. * ;

public class Test extends JFrame {
    private JTextArea textArea = new JTextArea ();
    private Document document = textArea.getDocument ();
    private DefaultEditorKit kit = new DefaultEditorKit ();
    private Action saveAction = new AbstractAction () {
        public void actionPerformed (ActionEvent e) {
            String s = JOptionPane.showInputDialog (
                Test.this,
                "Enter Filename:");
            if (s != null) {
                try {
                    kit.write (new FileWriter (s),
                        document, 0,
                        document.getLength ());
                }
                catch (Exception ex) {
                    ex.printStackTrace ();
                }
            }
        }
    };

    public Test () {
        Container contentPane = getContentPane ();

        try {
            kit.read (new FileReader ("Test.java"), document, 0);
        }
        catch (Exception ex) {
            ex.printStackTrace ();
        }

        final JTextArea status = new JTextArea ();
        JPanel p = new JPanel ();
        JSplitPane sp = new JSplitPane (JSplitPane.VERTICAL_SPLIT,
            p, status);
        sp.setDividerLocation (200);

        saveAction.putValue (Action.NAME, "Save ...");
        saveAction.setEnabled (false);

        p.setLayout (new BorderLayout ());
        p.add (new JScrollPane (textArea), BorderLayout.CENTER);
        contentPane.add (sp, BorderLayout.CENTER);

        textArea.getDocument ().addDocumentListener (
            new DocumentListener () {
                public void insertUpdate (DocumentEvent e) {
                    saveAction.setEnabled (true);
                    updateStatus (e);
                }
                public void removeUpdate (DocumentEvent e) {
                    saveAction.setEnabled (true);
                    updateStatus (e);
                }
                public void changedUpdate (DocumentEvent e) {

```



```

        saveAction.setEnabled (true);
        updateStatus (e);
    }
    private void updateStatus (DocumentEvent e) {
        status.append (e.getType () .toString ());
        status.append (" Offset: " + e.getOffset ());
        status.append (" Length: " + e.getLength () + "\n");
    }
}
setJMenuBar (createMenuBar ());
private JMenuBar createMenuBar () {
    JMenuBar menuBar = new JMenuBar ();
    JMenu editMenu = new JMenu ("Edit");

    editMenu.add (new DefaultEditorKit.CutAction ());
    editMenu.add (new DefaultEditorKit.CopyAction ());
    editMenu.add (new DefaultEditorKit.PasteAction ());
    editMenu.addSeparator ();
    editMenu.add (saveAction);

    menuBar.add (editMenu);
    return menuBar;
}

public static void main (String args []) {
    GJApp.launch (new Test (),
        "Using Document Listeners",
        300, 300, 650, 500);
}

```

21.5 加字符与加重器

所有的文本组件都有一个加字符与加重器，下面分别进行讨论。

21.5.1 加字符

一个加字符代表在文本组件的视图可以插入内容的位置。接口总结 21-5 总结了 Caret 接口。

接口总结 21-5 Caret

1. 变化监听器

```

public abstract void addChangeListener (ChangeListener)
public abstract void removeChangeListener (ChangeListener)

```

加字符支持变化监听器，无论加字符的位置在什么时候改变，都会通知变化监听器。可以使用上面列出的方法，给加字符添加变化监听器或删除加字符的变化监听器。

2. 安装

```

public abstract void install (JTextComponent)
public abstract void deinstall (JTextComponent)

```

在安装和删除文本组件的 UI 代表时，将调用上面列出的安装和删除方法。安装方法通常获取对文本组件文档的一个引用，从而可以给文档添加监听器。删除方法撤消安装方法所做的

一切，从而当文本组件的 UI 代表删除时，文本组件能恢复它的初始状态。

3. 标记、点和闪烁频率

```
public abstract int getBlinkRate ()
public abstract int getDot ()
public abstract int getMark ()
public abstract void moveDot ()
public abstract void setBlinkRate (int)
public abstract void setDot (int)
```

加字符维护一个点和标志。点代表加字符的当前位置，在没有执行选取时，点和标志是相同的。当文本被选中时，标志代表选取的初始位置，点则代表选取的结束位置。

加字符有一个闪烁频率，通过 `getBlinkRate` 和 `setBlinkRate` 方法可以访问加字符的闪烁频率。

4. 绘制/可视性/选取

```
public abstract boolean isVisible ()
public abstract void setVisible (boolean)
public abstract void setSelectionVisible (boolean)
public abstract boolean isSelectionVisible ()
public abstract void paint (Graphics)
```

使用上面列出的方法，可以设置加字符的可视性，以及使用加字符所作选取的可视性。`paint` 方法绘制加字符。

5. 魔术加字符位置

```
public abstract Point getMagicCaretPosition ()
public abstract void setMagicCaretPosition (Point)
```

魔术加字符位置维护文本中长度不等行的结束位置。例如，当加字符位于文档某一较长行的结束位置时，按向上箭头（↑）键，加字符将移到上面较短的行，此时加字符将置于较短行的结束位置。

21.5.2 加字符监听器

正如在接口总结 21-5 中所讨论的，加字符支持变化监听器，无论加字符的位置在什么时候改变，都会通知变化监听器。加字符监听器实现 `javax.swing.event` 包中的 `CaretListener` 接口，接口总结 21-6 总结了 `CaretListener` 接口。

接口总结 21-6 `CaretListener`

```
public abstract void caretUpdate (CaretEvent)
```

`CaretListener` 接口定义了一个方法，其输入参数为 `CaretEvent` 的一个实例。`CaretEvent` 类可以在 `javax.swing.event` 包找到，类总结 21-3 总结了 `CaretEvent` 类。

类总结 21-3 `CaretEvent`

扩展：`java.util.EventObject`

1. 构造方法：`public CaretEvent (Object source)`

用事件源来构造加字符事件，这个加字符事件是包含加字符的文本组件。

2. 方法

```
public abstract int getDot ()
```

```
public abstract int getMark ()
```

加字符事件提供方法，这些方法返回与位置被修改的加字符相关联的点和标志。

图 21-9 显示的小应用程序包含一个文本域，该文本域有一个加字符监听器，当激发加字符事件时，监听器将更新小应用程序的状态区。

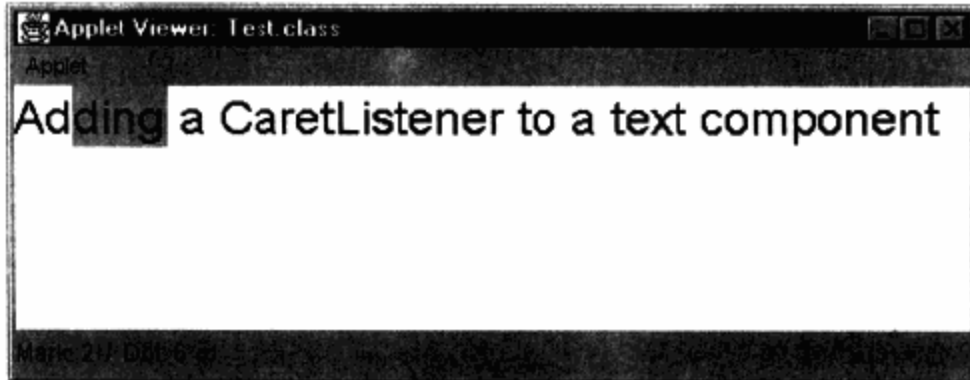


图 21-9 加字符监听器

例 21-6 列出了图 21-9 中显示的小应用程序的代码。

例 21-6 实现一个加字符监听器

```
import java.awt.* ;
import javax.swing.* ;
import javax.swing.event.* ;
import javax.swing.text.* ;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane () ;
        JTextArea textArea = new JTextArea () ;

        contentPane.add (textArea, BorderLayout.CENTER);
        textArea.setFont (new Font ("Dialog", Font.PLAIN, 24));
        textArea.addCaretListener (new CaretListener () {
            public void caretUpdate (CaretEvent e) {
                showStatus ("Mark: " + e.getMark () +
                           " / Dot: " + e.getDot ());
            }
        });
    }
}
```

这个小应用程序调用文本域的 `addCaretListener` 方法来添加加字符监听器，然后加字符监听器调用小应用程序的 `showStatus` 方法。

21.5.3 定制加字符

`JTextComponent` 类提供了一个 `setCaret` 方法，从而使 swing 文本组件可以设置加字符。Swing 也提供了一个 `DefaultCaret` 类，它扩展 `java.awt.Rectangle` 并实现 `Caret` 接口。`DefaultCaret` 类能够扩展从而实现定制加字符。

图 21-10 显示的小应用程序有一个文本域，该文本域配备了一个定制有加字符。在文本的底线定制有加字符显示为一个三角形。例 21-7 列出了图 21-10 所示小应用程序的代码。

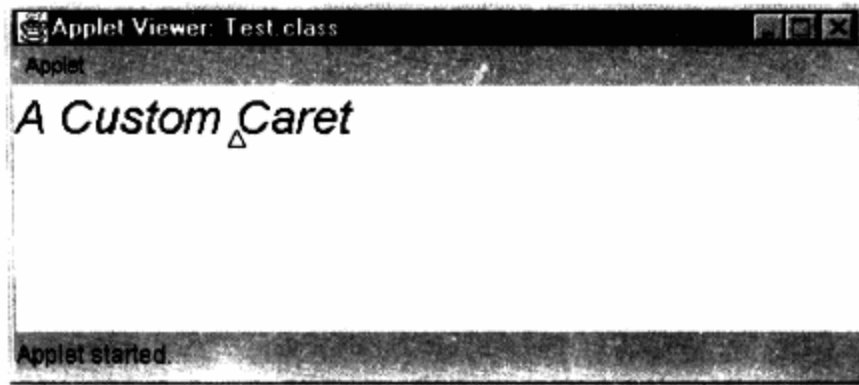


图 21-10 定制加字符
例 21-7 实现一个定制加字符

```
import javax.swing.*.*;
import javax.swing.plaf.*;
import javax.swing.text.*;
import java.awt.*.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JTextArea textArea = new JTextArea ();

        textArea.setCaret (new TriangleCaret (8));
        textArea.setFont (new Font ("Dialog", Font.ITALIC, 24));
        contentPane.add (textArea, BorderLayout.CENTER);
    }
}

class TriangleCaret extends DefaultCaret {
    private int triangleWidth, left, right, top, bottom, middle;

    public TriangleCaret (int triangleWidth) {
        this.triangleWidth = triangleWidth;
    }

    public void paint (Graphics g) {
        if (isVisible ()) {
            try {
                JTextComponent comp = getComponent ();
                Rectangle r = comp.modelToView (getDot ());
                setLocations (r);
                g.setColor (comp.getCaretColor ());

                g.drawLine (left, bottom, middle, top);
                g.drawLine (middle, top, right, bottom);
                g.drawLine (right, bottom, left, bottom);
            }
            catch (BadLocationException ex) {
                ex.printStackTrace ();
            }
        }
    }

    protected synchronized void damage (Rectangle r) {
        if (r != null) {
            setLocations (r);
        }
    }
}
```

```

        x = left;
        y = top;
        width = right - left + 1;
        height = bottom - top + 1;
    }
    private void setLocations (Rectangle r) {
        left = r.x - triangleWidth/2;
        right = r.x + triangleWidth/2;
        bottom = r.y + r.height - 1;
        top = bottom - triangleWidth;
        middle = r.x;
        repaint ();
    }
}

```

TriangleCaret 类扩展 DefaultCaret 并重载 paint 和 damage 方法。paint 方法使用 DefaultCaret.getComponent 方法来获得对包含加字符的文本组件的一个引用。然后使用该组件将组件视图中加字符的点转化为相应的矩形。接着设置组件加字符的颜色。最后使用 paint 方法在 setLocations 方法计算出来的位置绘制一个三角形。

Damage 方法负责设置加字符的位置和范围。要确保重载了 paint 方法的定制加字符也重载了 damage 方法，否则加字符移动时，将不能正确地绘制。

21.5.4 加重器

使用加重器可以增亮显示文本组件中的一个区域的文本，可以用增亮显示文本来表示文本被选取。当然增亮显示文本也可以用于其他目的，如增亮显示有拼写错误的单词。加重器在 Highlighter 接口中定义。接口总结 21-7 总结了 Highlighter 接口。

接口总结 21-7 Highlighter

1. 安装

```

public abstract void install (JTextComponent)
public abstract void deinstall (JTextComponent)

```

跟加字符一样，在安装和删除文本组件的 UI 代表时，将安装和删除加重器。缺省情况下，加重器保留对文本域的一个引用。传递该引用给 install 方法，从而可以访问组件文档和 UI 代表。

2. 绘制

```

public abstract void paint (Graphics)

```

加重器必须实现 paint 方法，这个方法绘制任何添加到加重器的增亮区。

3. 增亮区

```

public abstract Object addHighlight (int startPosition, int endPosition,
                                     Highlighter.HighlightPainter) throws BadLocationException
public abstract Highlighter.Highlight [] getHighlights ()
public abstract void changeHighlight (Object tag, int startPosition, int endPosition,) throws
                                     BadLocationException
public abstract void removeAllHighlights ()
public abstract void removeHighlight (Object)

```

Highlighter 接口中定义添加、改变和删除增亮区的方法。使用 `changeHighlight` 方法可以立即删除加重器的所有增亮区。

DefaultCaret 类使用加重器来增亮显示选取。使用 `DefaultCaret.getSelectionPainter` 方法，可以访问缺省的加字符加重器，图 21-11 示出的小应用程序有一个文本域，该文本域配备了一个定制加字符，这个定制加字符实现了一个定制的增亮区绘制器。

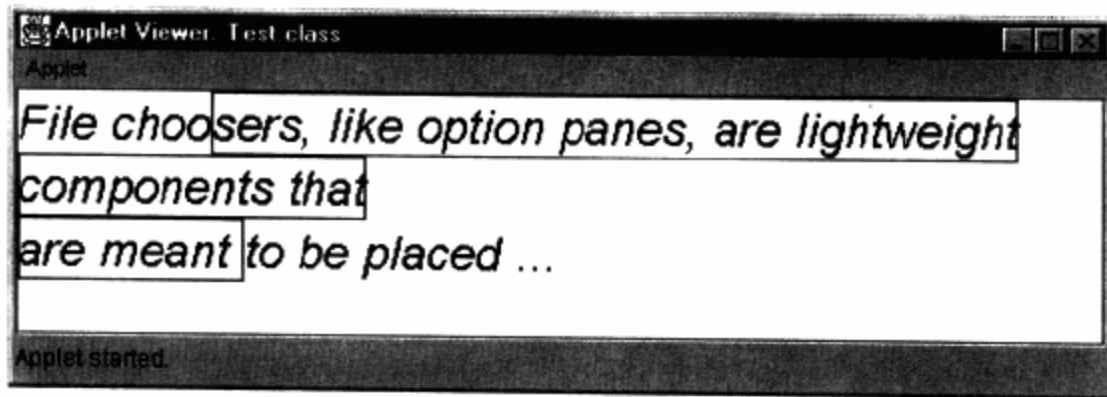


图 21-11 一个定制的加重器

例 21-8 列出了图 21-11 所示的小应用程序的代码。

例 21-8 一个定制的加重器

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;

public class Test extends JApplet {
    public Test () {
        Container contentPane = getContentPane ();
        JTextArea textArea = new JTextArea (
            "File choosers, like option panes, are lightweight \n" +
            "components that \n" +
            "are meant to be placed ...");
        textArea.setCaret (new BoxHighlightingCaret ());
        textArea.setFont (new Font ("Dialog", Font.ITALIC, 24));
        contentPane.add (new JScrollPane (textArea),
            BorderLayout.CENTER);
    }
}

class BoxHighlightingCaret extends DefaultCaret {
    private static BoxHighlighterPainter painter =
        new BoxHighlighterPainter (null);

    public Highlighter.HighlightPainter getSelectionPainter () {
        return painter;
    }
}

static class BoxHighlighterPainter
    extends DefaultHighlighter.DefaultHighlightPainter {
    private Color color;

    public BoxHighlighterPainter (Color color) {
        super (color);
        this.color = color;
    }
}
```

```

    }
    public Shape paintLayer (Graphics g, int p0, int p1,
                           Shape shape, JTextComponent comp,
                           View view) {
        Rectangle b = shape.getBounds ();
        try {
            g.setColor (getColor (comp));
            Rectangle r1 = comp.modelToView (p0);
            Rectangle r2 = comp.modelToView (p1);
            g.drawRect (r1.x, r1.y,          // x, y
                       r2.x - r1.x - 1, // width
                       r1.height - 1); // height
        }
        catch (BadLocationException ex) {
            ex.printStackTrace ();
        }
        return b;
    }
    private Color getColor (JTextComponent comp) {
        return color != null ? color :
            comp.getSelectionColor ();
    }
}

```

这个小应用程序把文本域的加字符设置为 `BoxHighlightingCaret` 的一个实例。`BoxHighlightingCaret` 扩展 `DefaultCaret` 并重载 `getSelectionPainter` 方法以便返回一个定制的增亮区绘制器。

`BoxHighlighterPainter` 扩展 `DefaultHighlighter.DefaultHighlighterPainter` 并重载 `paintLayer` 方法。在与增亮区绘制器相关联的文本组件中的每一行文本都调用 `paintLayer` 方法。`paintLayer` 方法的输入参数中有一个图形，可以在这个图形中进行绘制。输入参数还包括文本组件文档中要绘制区域的位置信息。

`BoxHighlighterPainter.paintLayer` 方法根据位置信息，在文本组件的视图中建立一个矩形，然后用组件的选取颜色绘制该矩形。

21.6 撤销/恢复

Swing 文本组件不仅激发在 21.4.2 “文档监听器”中所讨论的文档事件，而且激发可撤销的编辑事件。Swing 的撤销功能已在 6.7 “撤销/重复”中进行了讨论。图 21-12 中显示的小应用程序有一个文本域，该文本域配备有一个可撤销编辑监听器，从而使文档的可撤销编辑能够撤销和恢复。

图 21-12 中左上图显示的是这个小应用程序的初始状态，右上图显示的是文本域中的一些内容被剪切到剪贴板后这个小应用程序的状态。中图显示的是撤销剪切后这个小应用程序的状态，下图显示的是恢复剪切后这个小应用程序的状态。

这个小应用程序创建一个文本域，然后给文本域的文档添加了一个可撤销编辑监听器。这个小应用程序还创建了一个 `UndoManager` 实例和两个动作。这两个动作用于撤销和恢复文本域文档最近一次可撤销编辑。

添加到文本域的可撤销编辑监听器往撤销管理器添加可撤销编辑，并更新撤销和恢复动作。


```
public class Test extends JApplet {
    private JTextArea textArea = new JTextArea ("some content");
    private Document document = textArea.getDocument ();
    private UndoManager undoManager = new UndoManager ();
    private UndoLastAction undoAction = new UndoLastAction ();
    private RedoAction redoAction = new RedoAction ();
    ...
    public Test () {
        document.addUndoableEditListener (
            new UndoableEditListener () {
                public void undoableEditHappened (UndoableEditEvent e) {
                    undoManager.addEdit (e.getEdit ());
                    undoAction.update ();
                    redoAction.update ();
                }
            }
        );
    }
    ...
}
```

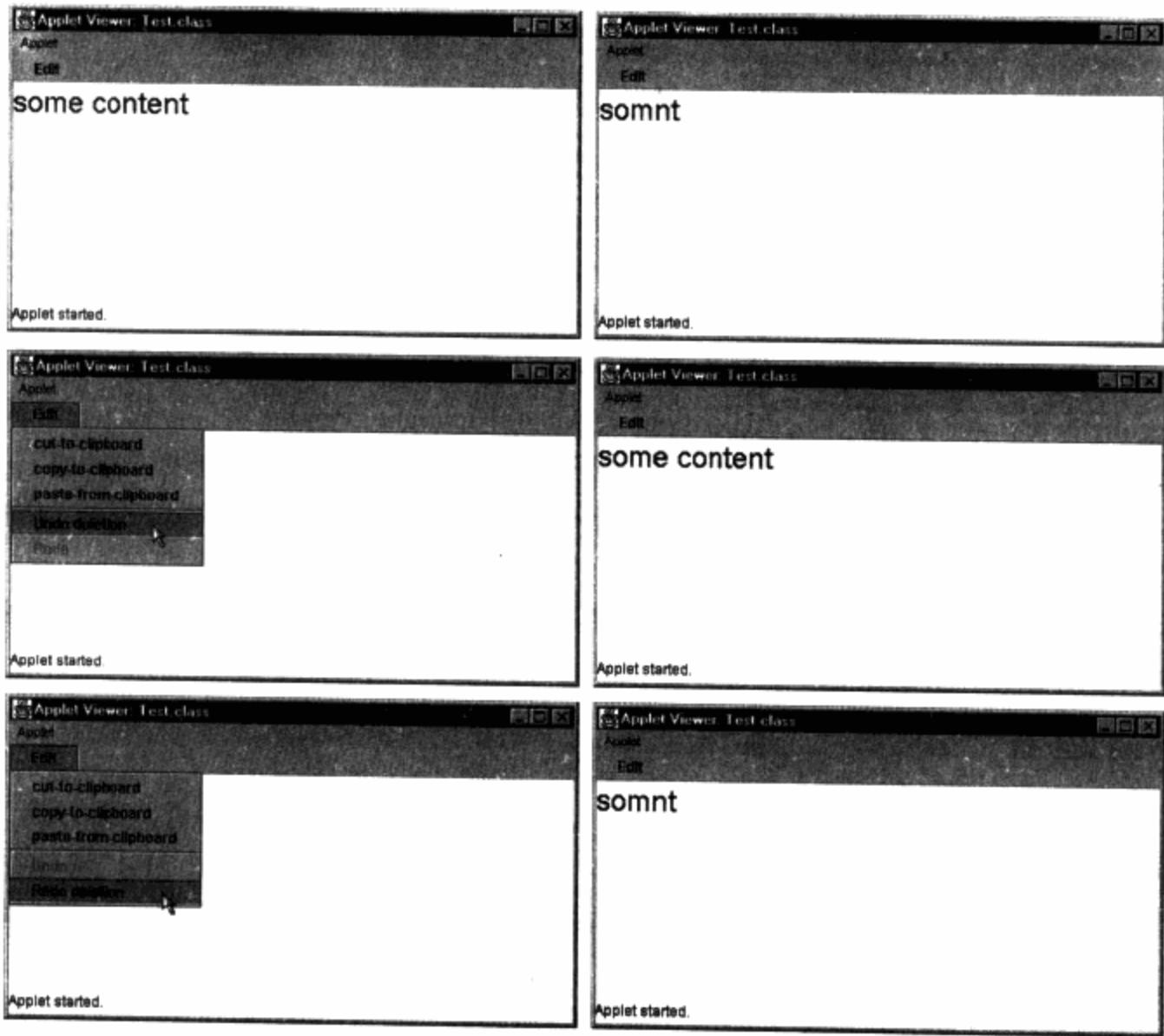


图 21-12 实现撤销/恢复

UndoLastAction 和 RedoLastAction 类扩展 AbstractAction 类并分别调用了 UndoManager.undo () 和 UndoManager.redo () 方法。当撤销和恢复文本域文档最近一次可撤销编辑后，每一个动作将更新另一个动作，从而设置动作的有效状态和名字。

```
...
class RedoAction extends AbstractAction {
    public RedoAction () {
```

```

        super ("Redo");
        update ();
    }

    public void actionPerformed (ActionEvent e) {
        undoManager.redo ();
        undoAction.update ();
        update ();
    }

    public void update () {
        boolean canRedo = undoManager.canRedo ();
        if (canRedo) {
            setEnabled (true);
            putValue (Action.NAME,
                undoManager.getRedoPresentationName ());
        }
        else {
            setEnabled (false);
            putValue (Action.NAME, "Redo");
        }
    }
}

class UndoLastAction extends AbstractAction {
    public UndoLastAction () {
        super ("Undo");
        update ();
    }

    public void actionPerformed (ActionEvent e) {
        undoManager.undo ();
        redoAction.update ();
        update ();
    }

    public void update () {
        boolean canUndo = undoManager.canUndo ();
        if (canUndo) {
            setEnabled (true);
            putValue (Action.NAME,
                undoManager.getUndoPresentationName ());
        }
        else {
            setEnabled (false);
            putValue (Action.NAME, "Undo");
        }
    }
}
}

```

例 21-9 列出了图 21-12 所示小应用程序的完整代码。

例 21-9 一个撤销和恢复的实现

```

import java.awt. * ;
import java.awt.event. * ;
import javax.swing. * ;
import javax.swing.event. * ;

```

```

import javax.swing.text.*;
import javax.swing.undo.*;

public class Test extends JApplet {
    private JTextArea textArea = new JTextArea ("some content");
    private Document document = textArea.getDocument ();
    private UndoManager undoManager = new UndoManager ();
    private UndoLastAction undoAction = new UndoLastAction ();
    private RedoAction redoAction = new RedoAction ();

    public Test () {
        Container contentPane = getContentPane ();

        createMenu ();
        contentPane.add (textArea, BorderLayout.CENTER);

        textArea.setFont (new Font ("Dialog", Font.PLAIN, 24));
        document.addUndoableEditListener (
            new UndoableEditListener () {
                public void undoableEditHappened (UndoableEditEvent e) {
                    undoManager.addEdit (e.getEdit ());
                    undoAction.update ();
                    redoAction.update ();
                }
            }
        );
    }

    private void createMenu () {
        JMenuBar menuBar = new JMenuBar ();
        JMenu editMenu = new JMenu ("Edit");

        editMenu.add (new DefaultEditorKit.CutAction ());
        editMenu.add (new DefaultEditorKit.CopyAction ());
        editMenu.add (new DefaultEditorKit.PasteAction ());

        editMenu.addSeparator ();

        editMenu.add (undoAction);
        editMenu.add (redoAction);

        menuBar.add (editMenu);
        setJMenuBar (menuBar);
    }

    class RedoAction extends AbstractAction {
        public RedoAction () {
            super ("Redo");
            update ();
        }

        public void actionPerformed (ActionEvent e) {
            undoManager.redo ();
            undoAction.update ();
            update ();
        }

        public void update () {
            boolean canRedo = undoManager.canRedo ();

            if (canRedo) {
                setEnabled (true);
                putValue (Action.NAME,
                    undoManager.getRedoPresentationName ());
            }
            else {

```

```

        setEnabled (false);
        putValue (Action.NAME, "Redo");
    }
}

class UndoLastAction extends AbstractAction {
    public UndoLastAction () {
        super ("Undo");
        update ();
    }

    public void actionPerformed (ActionEvent e) {
        undoManager.undo ();
        redoAction.update ();
        update ();
    }

    public void update () {
        boolean canUndo = undoManager.canUndo ();

        if (canUndo) {
            setEnabled (true);
            putValue (Action.NAME,
                undoManager.getUndoPresentationName ());
        }
        else {
            setEnabled (false);
            putValue (Action.NAME, "Undo");
        }
    }
}

```

21.7 JTextComponent

JTextComponent 类是所有 swing 组件的最终超类。组件总结 21-1 总结了 JTextComponent 类。

组件总结 21-1 JTextComponent

模型: javax.swing.text.Document
 UI 代表: javax.swing.plaf.basic.BasicTextUI
 绘制器: ——
 编辑器: ——
 激发事件: ——
 替代: ——
 类图: 见图 21-13

JTextComponent 类扩展 JComponent 并实现 Accessible 接口和 Scrollable 接口。JTextComponent 维护对它的文档、键映射、加字符和加重器的 private 引用。JTextComponent 的实例还跟踪用于绘制选取的文本颜色和文本组件加字符的颜色。

JComponent 的实例还维护对一个字符和一个动作的 private 引用，这个字符用于把焦点传递给这个文本组件，在键入该字符时将执行这个动作。

可变的加字符事件是 JTextComponent.MutableCaretEvent 类的一个实例，这个实例实现了 MouseListener 接口和 FocusListener 接口。当文本组件获得焦点时，可变的加字符事件保存对这

个组件的一个引用。当在这个组件中发生鼠标按下事件时，这个可变的加字符事件更新这个组件的加字符并激发一个加字符更新事件，把它自己指定为这个事件。

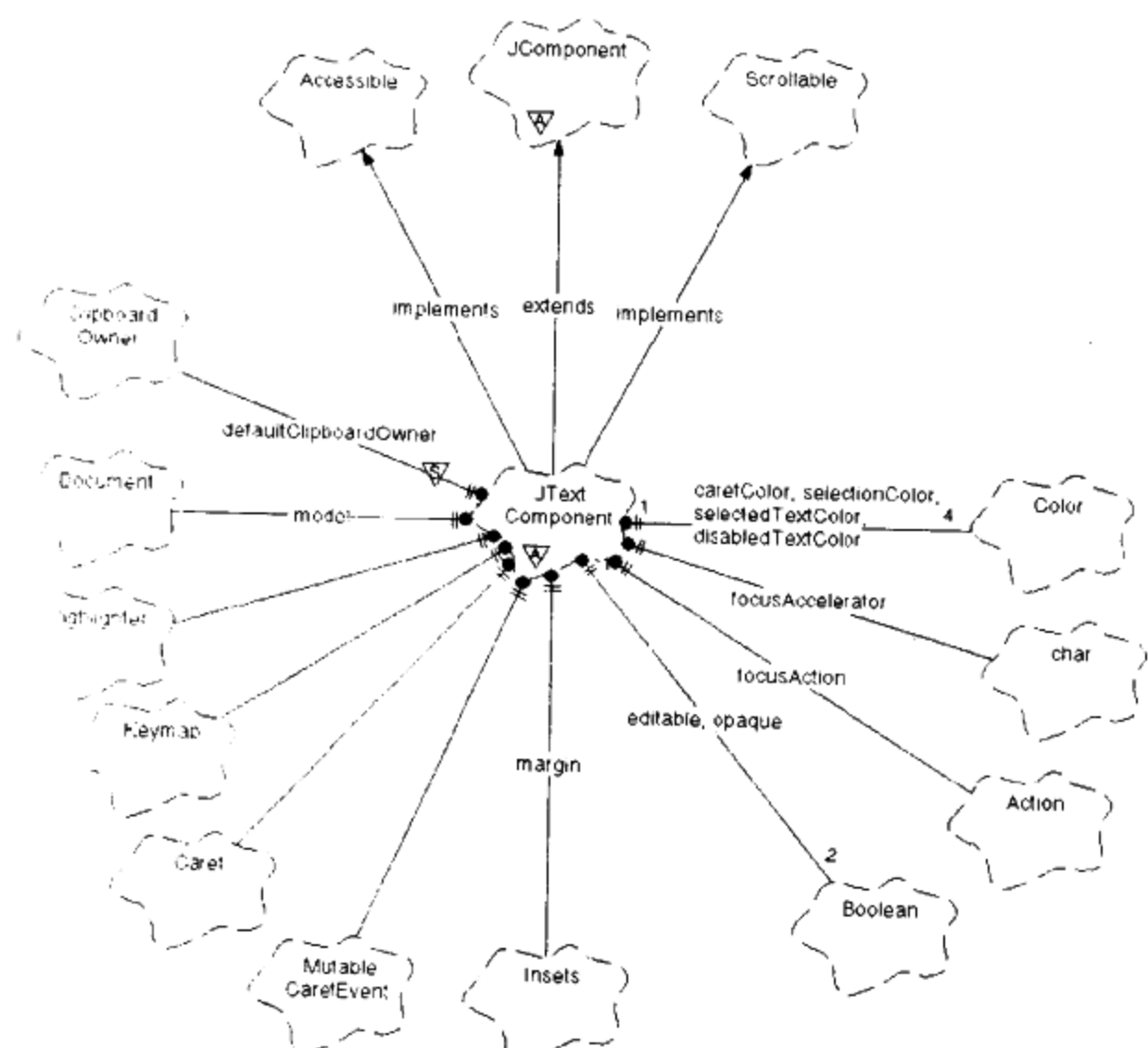


图 21-13 java.awt.TextComponent

JTextComponent 属性

表 21-4 列出了由 `JtextComponent` 类维护的属性。

表 21-4 JTextComponent 的属性

属性名字	数据类型	属性类型 ^①	访问 ^②	缺省 ^③
actions	Action []	S	G	L&F
caret	Caret	B	SG	BasicCaret
caretColor	Color	B	SG	L&F
caretPosition	int	S	SG	0
disabledTextColor	Color	B	SG	L&F
document	Document	B	SG	L&F
focusAccelerator	char	B	SG	—
highlighter	Highlighter	B	SG	L&F
keymap	Keymap	B	SG	L&F
margin	Insets	B	SG	L&F
selectedText	String	S	G	—
selectedTextColor	Color	B	SG	L&F
selectionColor	Color	B	SG	L&F
selectionEnd	int	S	SG	—
selectionStart	int	S	SG	—
text	String	S	SG	—

① B = 关联的 (激发 `PropertyChangeEvent`) / C = 受约束的 / I = 索引的 / S = 简单的 / Ch = 激发 `ChangeEvent`
② C = 可在创建时设置 / G = 获取方法 / S = 设置方法
③ L&F = 与界面样式有关

actions——一个文本组件可以执行的动作集。这些动作由一个文本组件的编辑器工具包来指定。动作可以与键相关联并用于菜单条和工具条中。

caret——文本组件的光标。加字符由文本组件的 UI 代表来初始化，但是我们可以定制加字符。在 21.5.3 节“定制加字符”中可以找到定制加字符的例子。

caretColor——文本组件的加字符的颜色。caretColor 属性是传递给文本组件的加字符的属性。

caretPosition——文本组件中加字符的位置。caretPosition 属性是传递给文本组件的加字符的属性。

disabledTextColor——绘制无效文本的颜色。

document——文本组件中使用的文档。文档存储内容并可以维护这个内容的属性。文档在 Document 接口中定义。在 21.4 节“文档”中可以找到有关文档的更多信息。

focusAccelerator——一个键，当按下这个键时，会使焦点跳到一个文本组件上。当与文本组件在同一个窗口的其他组件拥有焦点时，必须按该键，使文本组件获得焦点。

highLighter——所有的文本组件都支持多个增亮区。这些增亮区是由组件的加重器绘制的。缺省情况下，文本组件的 UI 代表创建组件中的加重器，但是我们也可以定制加重器。在 21.5.4 节“加重器”可以找到更多有关的信息。

keymap——每个文本组件都有一个键映射，从而使击键与操作对应。JTextComponent 类提供了一些方法来处理键映射。在 21.3 节“键映射”中可以找到更多的有关信息。

margin——一个边衬，它指定文本组件的边框和这个组件的内容之间的间隙。

selectedText——一个只读属性，返回文本组件中被选取的文本。使用加字符的点和标记从组件文档中获得该字符串。在 21.5.1 节“加字符”中可以找到更多的信息。

selectedTextColor——用来绘制选取文本的颜色。如果 JTextComponent.setSelectedTextColor() 的输入参数为 null，缺省的颜色是黑色。

selectionColor——选取的背景颜色。如果 JTextComponent.setSelectionColor() 的输入参数为 null，缺省颜色是白色。

selectionEnd——文档中当前选取的结束位置。使用 setSelectionEnd 方法可以设置 selectionEnd 的属性，从而调整组件中的加字符。

selectionStart——文档中当前选取的开始位置。使用 setSelectionStart 方法可以设置 selectionStart 的属性，从而调整组件中的加字符。

类总结 21-4 总结了 JTextComponent 类。

类总结 21-4 JTextComponent

扩展：JComponent

实现：javax.accessibility.Accessible, Scrollable

1. 常量

public static final String DEFAULT_KEYMAP

public static final String FOCUS_ACCELERATOR_KEY

DEFAULT_KEYMAP 字符串代表 static JTextComponent.getKeymap(String) 方法可以访问的缺省键映射。

FOCUS_ACCELERATOR_KEY 字符串代表关联属性 focusAccelerator 的名字。在 21.7 节中的“JTextComponent 属性”中可以找到关于 focusAccelerator 更多的信息。注意 KEY 在使用时是与 PROPERTY 等价的。绝大多数的 swing 组件使用它来表示关联属性的名字。例如，JTree 的选取

模型由 `JTree.SELECTION_MODEL_PROPERTY` 来表示。

2. 构造方法

```
public JTextComponent ()
```

这个无参数构造方法是 `JTextComponent` 类提供的唯一的一个构造方法。这个构造方法将组件的布局管理器设置为 `null`，并将从 `JComponent` 继承的 `editable` 属性设置为真。组件的视图负责布局文本组件内容。有关文本组件视图的更多信息，请参见第 23 章的内容。这个构造方法还把鼠标和焦点监听器添加到激发加字符事件的组件中。

3. 方法

(1) 文档/文本

```
public Document getDocument ()
public void setDocument (Document)
public String getText ()
public String getText (int offset, int length) throws BadLocationException
public void setText (String)
public Color getDisabledTextColor ()
public void setDisabledTextColor (Color)
```

使用上面列出的方法，可以访问文本组件的文档、文本以及无效文本颜色。`setDocument` 方法激发属性修改事件，通过为文档创建和安装一个视图来处理该事件。

通过 `getText` 方法可以获取文本组件中的文本。通过指明文本在文档中的起始位置以及要读取的文本的长度，可以访问一个区域内的文本。如果所指明的起始位置或长度无效，`getText` 方法将弹出 `BadLocationException` 异常信息。

(2) 动作

```
public Action [] getActions ()
```

`getActions` 方法将返回一个动作数组，这个数组由一个文本组件来执行。动作是文本组件实现基本功能的主要机制。在菜单和工具条中，动作是很有用的，它还用在键映射中。

(3) 键映射

```
public static Keymap addKeymap (String name, Keymap)
public static Keymap getKeymap (String name)
public static void removeKeymap (String name)
public static void loadKeymap (Keymap, JTextComponent.KeyBinding [], Action [])
public void setKeymap (Keymap)
public Keymap getKeymap ()
```

键映射是击键与操作的捆绑集。每个文本组件都有一个键映射。如果给 `JTextComponent.setKeymap []` 传递一个 `null` 引用，则文本组件将不接受键盘输入。

使用 `JTextComponent.setKeymap` 方法可以设置键映射。因为键映射可以包含其他的键映射，而且因为键映射分层地进行解析，所以上面列出的这些静态方法通常用于修改一个文本的键映射。有关键映射的更多信息，请参见在 21.3 节“键映射”。

(4) 加字符/登记加字符监听器

```
public void addCaretListener (CaretListener)
public void removeCaretListener (CaretListener)
protected void fireCaretUpdate (CaretEvent)
public Caret getCaret ()
public Color getCaretColor ()
public int getCaretPosition ()
public void setCaret (Caret)
```



```
public void setCaretColor (Color)
public void setCaretPosition (int position)
public void moveCaretPosition (int position)
```

上面列出的方法可以用来操纵一个文本组件的加字符并登记加字符监听器。有关加字符和加字符监听器的详细内容，请参见 21.5 节“加字符和加重器”。

setCaretPosition 和 moveCaretPosition 方法分别调用了 Caret.setDot 和 Caret.moveDot。前者只是简单修改加字符的位置，后者则将加字符从以前的位置移到一个新的位置从而构成一个选取。

(5) 加重器

```
public Highlighter getHighlighter ()
public void set tHighlighter (Highlighter)
```

使用上面列出的方法可以访问文本组件的加重器，在 21.5 节“加字符和加重器”中可以找到关于加重器的更多信息。

(6) 剪切/拷贝/粘贴

```
public void copy ()
public void cut ()
public void paste ()
```

使用上面列出的方法可以方便地将文本剪切、拷贝到系统剪切板中，以及把系统剪切板中的文本粘贴到文本组件。

拷贝方法将当前选取的文本从文本组件拷贝到剪切板。该方法并没有把文本从文本组件中移走。当没有选取文本时，调用该方法，该方法将不会做任何事。

剪切方法将当前选取的文本从文本组件移走，然后拷贝到剪切板。当没有选取文本时，调用该方法，该方法将不会做任何事。

粘贴方法拷贝系统剪切板中的当前选取，然后将它插入当前加字符的位置前。如果系统剪切板为 null，该方法将不会做任何事。

(7) 选取

```
public String getSelectedText ()
public Color getSelectedTextColor ()
public Color getSelectionColor ()
public int getSelectionEnd ()
public int getSelectionStart ()
public void setSelectionColor (Color)
public void setSelectionEnd (int)
public void setSelectionStart (int)
public void replaceSelection (String)
public void select (int start, int end)
public void selectAll ()
```

上面列出的是 JTextComponent 提供的用来控制选取的方法。前两个方法分别返回当前选取的文本以及绘制选取文本所用的颜色。如果调用 getSelectedText 方法时，没有选取文本，方法将返回一个 null 引用。

getSelectionColor 返回绘制选取文本所用的颜色。getSelectionStart 和 getSelectionEnd 方法分别返回当前选取的开始和结束位置。如果当前没有选取，则返回加字符点的位置。如果文档中没有文本，则返回 0。

replaceSelection 方法用指定的文本替换当前选取的文本。当该方法被调用时，如果当前没有选取文本，则把文本插入当前加字符的点的位置前。如果传递给该方法的文本为 null，则删

除当前选取。

`select` 方法与 AWT 的文本组件保持兼容。当指定的开始或结束位置无效时，该方法将不会做任何事。如果指定的开始和结束位置有效，该方法将调用 `setCaretPosition()`，然后调用 `moveCaretPosition()`。

(8) 读/写

```
public void read (Reader, Object) throws IOException
```

```
public void write (Writer) throws IOException
```

`read` 和 `write` 方法分别使用文本组件的编辑器工具包来创建和保存文档。

`read` 方法创建一个与这个组件的编辑器工具包相适应的模型，并用流中的数据来初始化该模型。`write` 方法将文档中的文本保存为纯文本。

(9) 视图/模型转换

```
public Rectangle modelToView (int position) throws BadLocationException
```

```
public int viewToModel (Point)
```

在上面列出的方法中，前者将模型的位置转换为视图的坐标，后者则将视图的坐标转换为模型的位置。这些方法交给这个文本组件的 UI 代表

(10) Scrollable 接口的实现

```
public Dimension getPreferredSize ()
```

```
public int getScrollableBlockIncrement (Rectangle, int, int)
```

```
public boolean getScrollableTracksViewportHeight ()
```

```
public boolean getScrollableTracksViewportWidth ()
```

```
public int getScrollableUnitIncrement (Rectangle, int, int)
```

上面列出的方法由 `Scrollable` 接口来定义。有关 `Scrollable` 接口的详细内容，请参见 13.3 节“`Scrollable` 接口”。

`getPreferredSize` 返回组件的首选大小。`getScrollableUnitIncrement` 方法根据传送给这个方法的滚动方向返回这个组件宽度或高度的十分之一。

`getScrollableBlockIncrement` 方法根据传送给这个方法的滚动方向返回这个组件可视区的宽度或高度。

`getScrollableTracksViewportHeight` 和 `getScrollableTracksViewportWidth` 方法决定文本域的宽度或高度是否跟踪组件视口的宽度或高度（当文本组件包含在视口中时）。如果组件视口的宽度或高度大于文本域的宽度或高度时，这两个方法将返回真，否则返回假。换句话说就是在缺省情况下，只有视口比组件大或相等时，文本组件的宽度和高度才与组件视口的宽度和高度相等。

(11) 杂项属性

```
public char getFocusAccelerator ()
```

```
public Insets getMargin ()
```

```
public boolean isEditable ()
```

```
public boolean isFocusTraversable ()
```

```
public boolean isOpaque ()
```

```
public void setEditable (boolean)
```

```
public void setEnabled (boolean)
```

```
public void setFocusAccelerator (char)
```

```
public void setMargin (Insets)
```

```
public void setOpaque (boolean)
```

使用上面的方法可以访问文本组件的属性。

(12) 杂项方法

```
protected String paramString ()  
protected void processComponentKeyEvent (KeyEvent)  
public void removeNotify ()
```

paramString 方法返回文本组件的文本描述。该方法通常仅用于调试，而且要确保方法返回非 null 字符串。

当一个键事件被传递给焦点管理器和已登记的键监听器时，processComponentKeyEvent 方法被调用。如果焦点管理器或键监听器没有定制该事件，processComponentKeyEvent 方法执行缺省的动作并定制该事件。

当一个组件从它的容器中移走时，则调用 removeNotify 方法。该方法删除对作为 focused-Component 的组件的引用。

(13) 可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext ()  
public TextUI getUI ()  
public void setUI (TextUI)  
public void updateUI ()
```

上面列出的方法可以在大多数 JComponent 扩展中找到。Swing 轻量组件能够返回它们的 UI 代表的类名及包含组件的可访问性信息的相关内容。updateUI 方法在组件配备了 UI 代表时调用。

21.8 本章回顾

Swing 文本无疑是 Swing 最复杂的部分。本章介绍了所有 Swing 文本组件都共享的基本功能，包括动作、键映射、文档、加字符、加重器、撤销和恢复。还介绍了 JTextComponent 类，它是所有文本组件的最终超类。本章特意没有涉及 Swing 文本的高级主题，如视图、元素、属性集、风格、风格的相关内容等。

第 22 章将分别介绍每一个 Swing 文本组件，然后再在第 23 章论述高级主题。

第 22 章 文本组件

Swing 提供两种完全不同的文本组件类型:简单文本控件和风格文本组件。简单文本控件一次只能显示一种字体和一种颜色,而风格文本组件则可以显示多种字体和颜色。前者有单行文本域(JTextField)、口令域(JPasswordField)和多行文本域(JTextArea),后者有编辑器窗格(JEditorPane)和文本窗格(JTextPane)。

单行文本域和多行文本域编辑纯文本是很有用的。JTextField 和 JTextArea 与它们的 AWT 对等组件(java.awt.TextField 和 java.awt.TextArea)在源代码上几乎是完全兼容的。

用编辑窗格和文本窗格可编辑不同风格文本来显示不同内容。JTextPane 类为在文本窗格中显示的内容提供了设置字符和段落属性的方法。文本窗格还能够嵌入组件和图标。JEditorPane 和 JTextPane 没有 AWT 对等组件。

22.1 JTextField

JTextField 组件显示单行可编辑文本,它只使用一种字体和一种颜色,可以为单行文本域的文本指定水平排列方式(LEFT、CENTER 或 RIGHT)[○]。

缺省情况下,当单行文本域有焦点时按下 Enter 键(KeyEvent.VK_ENTER)将引起这个文本域激发一个动作事件。这种行为与 AWT 的单行文本域兼容,但与 Swing 的缺省按钮不一致(Swing 的缺省按钮在按下 Enter 键时也被激活)。22.1.6“JTextField 事件”讨论了这个问题的解决方法。

通过指定单行文本域想要显示的列数可以设置单行文本域的首选宽度。单行文本域的实际宽度通常由布局管理器来设置,因此设置单行文本域的列数并不保证它的实际宽度。

图 22-1 所示的小应用程序说明了单行文本域的列和水平排列。

图 22-1 中的左上图显示了这个小应用程序初始的样子;缺省时,水平排列方式和列数分别为 JTextField.LEFT 和 0;右上图显示了在水平排列方式被设置为 JTextField.CENTER、列数被设置为 10 后这个小应用程序的样子。

图 22-1 的左下图显示了水平排列方式为 LEFT、列数为 5 的单行文本域的样子。注意,单行文本域可以容纳多于其列数的字符,因为列宽被设置成字符 m 的宽度,m 字符在大多数字体中是最宽的字符。

这个小应用程序用 JTextField 构造方法创建一个文本域,该构造方法以代表这个单行文本域初始内容的一个字符串为参数。把这个单行文本域放在一个面板上,指定这个面板为这个小应用程序内容窗格的中心组件。把这个单行文本域包裹在一个面板中,是因为这个面板有一个 FlowLayout 布局管理器,它可以根据组件的首选大小来布局组件。因为这个单行文本域将根据它的首选大小被布局,所以为这个单行文本域设置的列数将决定这个单行文本域的实际宽度。

```
public class Test extends JApplet {  
    private JPanel textFieldPanel = new JPanel();  
    private JTextField textField =
```

○ 这些常量被列在 JTextField 的公共常量中。

```

        new JTextField("initial content");
    public void init() {
        Container contentPane = getContentPane();

        textFieldPanel.add(textField);

        contentPane.setLayout(new BorderLayout(0,20));
        contentPane.add(new ControlPanel(), BorderLayout.NORTH);
        contentPane.add(textFieldPanel, BorderLayout.CENTER);
    }
    ...

```

`columns` 组合框有一个动作监听器,这个监听器设置这个单行文本域的列并使这个单行文本域重新有效。无论何时修改影响 Swing 组件外观的属性,这个组件都负责更新其显示。因此,在设置了这个单行文本域的列后,并不需要调用 `revalidate()`。

`alignments` 组合框有一个动作监听器,这个动作监听器根据组合框当前的选取来设置这个单行文本域的水平排列方式。

```

    ...
    class ControlPanel extends JPanel {
        private JComboBox alignments = new JComboBox();
        private JComboBox columns = new JComboBox();

        public ControlPanel() {
            ...
            columns.addActionListener(new ActionListener() {
                public void actionPerformed (ActionEvent e) {
                    Integer c =
                        (Integer)columns.getSelectedItem();

                    textField.setColumns(c.intValue());

                    // the following call to revalidate()
                    // should not be necessary
                    revalidate();

                    textField.setScrollOffset(0);
                }
            });
            alignments.addActionListener(new ActionListener() {
                public void actionPerformed (ActionEvent e) {
                    int index = alignments.getSelectedIndex();

                    if(index == 0)
                        textField.setHorizontalAlignment (
                            JTextField.LEFT);
                    else if(index == 1)
                        textField.setHorizontalAlignment (
                            JTextField.CENTER);
                    else if(index == 2)
                        textField.setHorizontalAlignment (
                            JTextField.RIGHT);
                }
            });
        }
    }
    ...

```

例 22-1 列出了图 22-1 所示的小应用程序的完整代码。

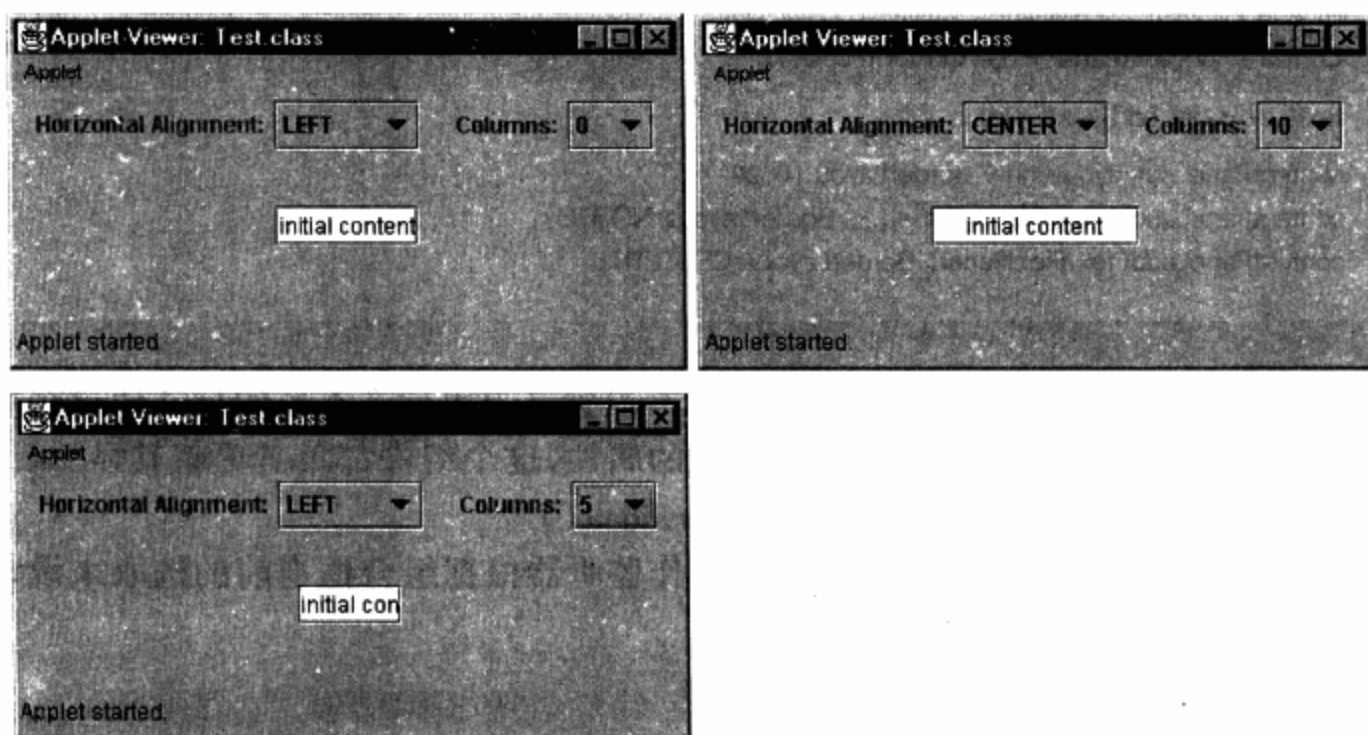


图 22-1 显式地设置一个单行文本域的列

例 22-1 单行文本域的排列方式和列数

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JApplet {
    private JPanel textFieldPanel = new JPanel();
    private JTextField textField =
        new JTextField("initial content");

    public void init() {
        Container contentPane = getContentPane();

        textFieldPanel.add(textField);
        contentPane.setLayout(new BorderLayout(0, 20));
        contentPane.add(new ControlPanel(), BorderLayout.NORTH);
        contentPane.add(textFieldPanel, BorderLayout.CENTER);
    }

    class ControlPanel extends JPanel {
        private JComboBox alignments = new JComboBox();
        private JComboBox columns = new JComboBox();

        public ControlPanel() {
            columns.addItem(new Integer(0));
            columns.addItem(new Integer(5));
            columns.addItem(new Integer(10));
            columns.addItem(new Integer(15));

            alignments.addItem("LEFT");
            alignments.addItem("CENTER");
            alignments.addItem("RIGHT");

            add(new JLabel("Horizontal Alignment:"));
            add(alignments);
            add(Box.createHorizontalStrut(10));
        }
    }
}

```

```

add(new JLabel("Columns:"));
add(columns);

columns.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Integer c =
            (Integer)columns.getSelectedItem();

        textField.setColumns(c.intValue());

        // the following call to revalidate()
        // should not be necessary
        revalidate();

        textField.setScrollOffset(0);
    }
});

alignments.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int index = alignments.getSelectedIndex();

        if(index == 0)
            textField.setHorizontalAlignment(
                JTextField.LEFT);
        else if(index == 1)
            textField.setHorizontalAlignment(
                JTextField.CENTER);
        else if(index == 2)
            textField.setHorizontalAlignment(
                JTextField.RIGHT);
    }
});
}
}
}

```

Swing 提示

单行文本域列

指定单行文本域的列数经常会产生预料不到的结果,因为人们很容易误解单行文本域的列的两个方面。

首先,单行文本域的列宽是单行文本域中当前字体下字符 **m** 的宽度。因为大多数字体有各种不同的宽度,而且通常 **m** 是最宽的字符,所以单行文本域的宽度经常比它所需要的宽度要稍宽一些。其次,设置单行文本域的列数只是设置了单行文本域的首选大小。单行文本域的实际大小是否与列数相符取决于单行文本域是否根据其首选大小来进行布局。

22.1.1 水平可视性和滚动偏移

JTextField 维护一个代表单行文本域水平可视性(horizontal visibility)的范围限定模型。范围限定模型包含有以下属性: value、minimum、maximum、extent。对单行文本域而言,这个域的可视性 value 值代表在这个域中显示的文本的滚动偏移。

图 22-2 示出的小应用程序阐明了单行文本域的水平可视性和滚动偏移。图 22-2 的上图显

示了这个小应用程序的初始状态。滚动偏移缺省值为 0, 即这个域的文本在这个域左上角的 x 坐标值为 0。

图 22-2 的中图示出了在用滑杆使这个单行文本域有滚动偏移后这个小应用程序的样子。此时, 文本在这个域的左上角的 x 坐标值是 179。

图 22-2 的下图示出了这个单行文本域的列被设置为 0 后小应用程序的样子, 在这种情况下, 这个单行文本域根据其首选宽度来调整大小。注意, 这个域的水平可视性的可见部分(可见部分定义可见部分的宽度)与这个域的增大尺寸相匹配。

这个小应用程序用一个 `JTextField` 的构造方法来创建一个文本域, 这个构造方法以表示这个单行文本域初始内容的一个字符串和这个单行文本域想要显示的列数为参数。

滚动偏移滑杆和列组合框都有两个监听器, 这两个监听器分别设置这个单行文本域的滚动偏移和列数。

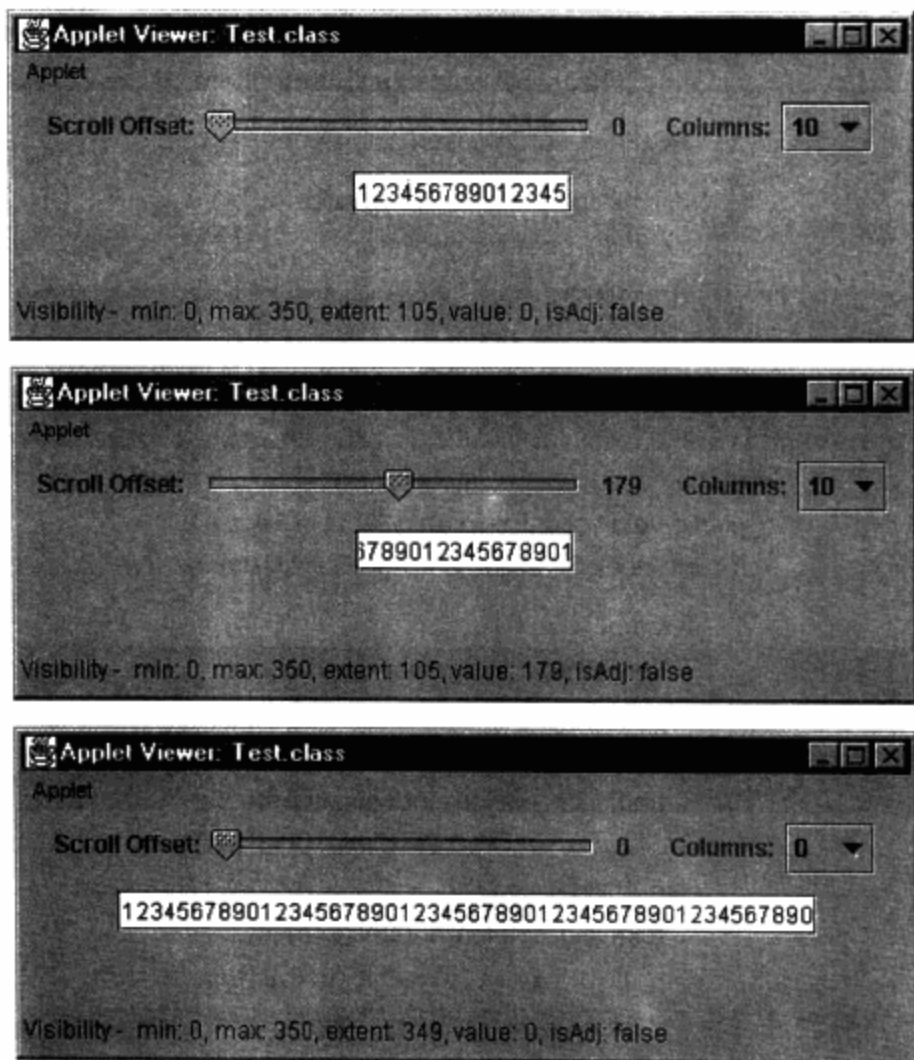


图 22-2 单行文本域的滚动偏移

```
public class Test extends JApplet {
    private JTextField textField = new JTextField (
        "12345678901234567890123456789012345678901234567890", 10);
    ...
    class ControlPanel extends JPanel {
        ...
        public ControlPanel() {
            ...
            slider.addChangeListener (new ChangeListener() {
                public void stateChanged (ChangeEvent e) {
                    textField.setScrollOffset(slider.getValue());

                    Integer i =
                        new Integer(textField.getScrollOffset());
                    BoundedRangeModel m =
                        textField.getHorizontalVisibility();

                    display.setText( i.toString());

                    showStatus ("Visibility-min:" +
                        m.getMinimum() +
                        ", max:" + m.getMaximum() +
                        ", extent:" + m.getExtent() +
                        ", value:" + m.getValue() +
```

```

        ", isAdj: " +
        m.getValuesAdjusting());
    }
    });
    columns.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            Integer c =
                (Integer)columns.getSelectedItem();

            textField.setColumns(c.intValue());

            // the following call to revalidate();
            // should not be necessary
            revalidate();

            textField.setScrollOffset(0);
        }
    });
}
}
}

```

例 22-2 列出了图 22-2 所示小应用程序的完整代码。

例 22-2 单行文本域的滚动偏移

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Test extends JApplet {
    private JTextField textField = new JTextField (
        "1234567890123456789012345678901234567890", 10);

    public void init() {
        Container contentPane = getContentPane();
        JPanel textFieldPanel = new JPanel();

        textFieldPanel.add(textField);

        contentPane.add(new ControlPanel(), BorderLayout.NORTH);
        contentPane.add(textFieldPanel, BorderLayout.CENTER);
    }

    class ControlPanel extends JPanel {
        private JLabel display = new JLabel("");
        private JSlider slider = new JSlider (
            textField.getHorizontalVisibility());
        private JComboBox columns = new JComboBox();

        public ControlPanel() {
            columns.addItem( new Integer(0));
            columns.addItem( new Integer(5));
            columns.addItem( new Integer(10));
            columns.addItem( new Integer(15));

            columns.setSelectedIndex(2);

            add( new JLabel ("Scroll Offset : "));
            add(slider);
            add(display);
        }
    }
}

```

```

add( Box.createHorizontalStrut(10));
add( new JLabel ("Columns : "));
add( columns);

slider.addChangeListener( new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        textField.setScrollOffset( slider.getValue());

        Integer i =
            new Integer( textField.getScrollOffset());
        BoundedRangeModel m =
            textField.getHorizontalVisibility();

        display.setText( i.toString());

        showStatus("Visibility-min: " +
            m.getMinimum() +
            ", max: " + m.getMaximum() +
            ", extent: " + m.getExtent() +
            ", value: " + m.getValue() +
            ", isAdj: " +
            m.getValueIsAdjusting());
    }
});

columns.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Integer c =
            (Integer) columns.getSelectedItem();
        textField.setColumns( c.intValue());

        // the following call to revalidate();
        // should not be necessary
        revalidate();

        textField.setScrollOffset(0);
    }
});
}
}
}

```

22.1.2 布局单行文本域

Swing 的单行文本域组件比较简单,易于使用,但很难正确地设置它们的尺寸和位置。本节给出了一个利用复杂的 GridBagLayout 布局管理器的简单例子。如图 22-3 所示,本例可用来布局标签/单行文本域对。

图 22-3 所示的小应用程序布局标签对和单行文本域并排列这些标签和单行文本域。下面的代码取自一个包含了标签和单行文本域的面板。这个面板的构造方法实例化 GridBagLayout 和 GridBagConstraints 的实例,并把 GridBagLayout 设置为这个容器的布局管理器。

```

...
GridBagLayout      gbl = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();

setLayout(gbl);
...

```

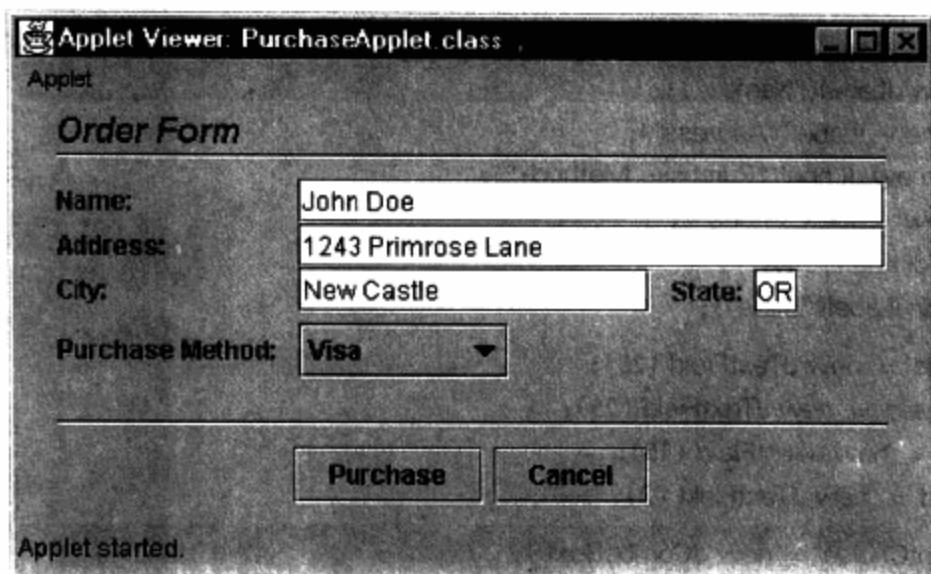


图 22-3 单行文本域和 GridBagLayout

接着,为这些标签和单行文本域设置约束条件时,出现了一个有趣的样式:因为这些标签被放置在这些单行文本域的左边,而且因为这些单行文本域填充它们格子的剩余部分,所以 `grid-width` 约束条件在 1(用于这些标签)和 `GridBagConstraints.REMAINDER`(用于这些单行文本域)之间切换。

```

...
gbc.anchor = GridBagConstraints.WEST;

gbc.gridwidth = 1;
gbc.insets = new Insets(0,0,0,0);
add(name, gbc);

add(Box.createHorizontalStrut(10));

gbc.gridwidth = GridBagConstraints.REMAINDER;
add(nameField, gbc);

gbc.gridwidth = 1;
add(address, gbc);

add(Box.createHorizontalStrut(10));

gbc.gridwidth = GridBagConstraints.REMAINDER;
add(addressField, gbc);

gbc.gridwidth = 1;
add(city, gbc);

```

例 22-3 列出了图 22-3 所示应用程序的完整代码。

例 22-3 用 GridBagLayout 来布局单行文本域

[illegible]

```
class ButtonPurchaseForm extends JPanel {
    JSeparator sep = new JSeparator();
    JLabel title = new JLabel("Order Form");
    JLabel name = new JLabel("Name:");
    JLabel address = new JLabel("Address");
    JLabel payment = new JLabel("Purchase Method:");
    JLabel phone = new JLabel("Phone");
    JLabel city = new JLabel("City:");
    JLabel state = new JLabel("State");

    JTextField nameField = new JTextField(25);
    JTextField addressField = new JTextField(25);
    JTextField cityField = new JTextField(15);
    JTextField stateField = new JTextField(2);

    JComboBox paymentChoice = new JComboBox();
    JButton paymentButton = new JButton("Purchase");
    JButton cancelButton = new JButton("Cancel");

    public ButtonPurchaseForm() {
        GridBagLayout gbl = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();

        setLayout(gbl);

        paymentChoice.addItem("Visa");
        paymentChoice.addItem("MasterCard");
        paymentChoice.addItem("COD");

        title.setFont(new Font("Times-Roman",
                               Font.BOLD + Font.ITALIC,
                               16));

        gbc.anchor = GridBagConstraints.NORTHWEST;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        add(title, gbc);

        gbc.anchor = GridBagConstraints.NORTH;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.insets = new Insets(0,0,10,0);
        add(sep, gbc);

        gbc.anchor = GridBagConstraints.WEST;
        gbc.gridwidth = 1;
        gbc.insets = new Insets(0,0,0,0);
        add(name, gbc);

        add(Box.createHorizontalStrut(10));
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        add(nameField, gbc);

        gbc.gridwidth = 1;
        add(address, gbc);

        add(Box.createHorizontalStrut(10));
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        add(addressField, gbc);

        gbc.gridwidth = 1;
        add(city, gbc);

        add(Box.createHorizontalStrut(10));
        add(cityField, gbc);
    }
}
```

```

add(Box.createHorizontalStrut(10));
add(state, gbc);
add(Box.createHorizontalStrut(5));

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.fill = GridBagConstraints.NONE;
add(stateField, gbc);

gbc.gridwidth = 1;
add(payment, gbc);

gbc.insets = new Insets(5,0,5,0);
add(Box.createHorizontalStrut(10));
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.fill = GridBagConstraints.NONE;
add(paymentChoice, gbc);

ButtonPanel buttonPanel = new buttonPanel();
buttonPanel.add(paymentButton);
buttonPanel.add(cancelButton);

gbc.anchor = GridBagConstraints.SOUTH;
gbc.insets = new Insets(15,0,0,0);
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridwidth = 7;
add(buttonPanel, gbc);
}

class ButtonPanel extends JPanel {
    JPanel buttonPanel = new JPanel();
    JSeparator separator = new JSeparator();

    public ButtonPanel() {
        buttonPanel.setLayout(
            new FlowLayout(FlowLayout.CENTER));
        setLayout(new BorderLayout(0, 5));
        add(separator, "North");
        add(buttonPanel, "Center");
    }

    public void add(JButton button) {
        buttonPanel.add(button);
    }
}

```

Swing 提示

关于利用 GridBagLayout 来布局标签/单行文本域对

GridBagLayout 是布局标签/单行文本域对的强大的布局管理器。标签/单行文本域对通常像图 22-3 中的姓名和地址那样排列, 所以 gridwidth 约束条件在 1 (用于标签) 和 GridBagConstraints.REMAINDER (用于单行文本域) 之间切换。

22.1.3 使单行文本域有效

单行文本域通常要使输入的数据有效; 例如, 在 21.4.1 节“定制文档”中给出了一个只允许

输入整数的例子。本节讨论一个更复杂的任务,它实现一个日期单行文本域,如图 22-4 所示。

图 22-4 所示的日期单行文本域是一个配备了定制文档的 `JTextField` 实例。如图 22-4 所示,文档监视文本的插入和删除。左上图显示了输入 0 和 2 之后这个域的样子;右上图显示了接着输入 1 之后的这个域的样子。注意输入 1 后加字符跳过了分隔符。下面的图显示了加字符在分隔符前面时按 `Backspace` 键的情形——加字符回退两个空格,从而不能删除分隔符。

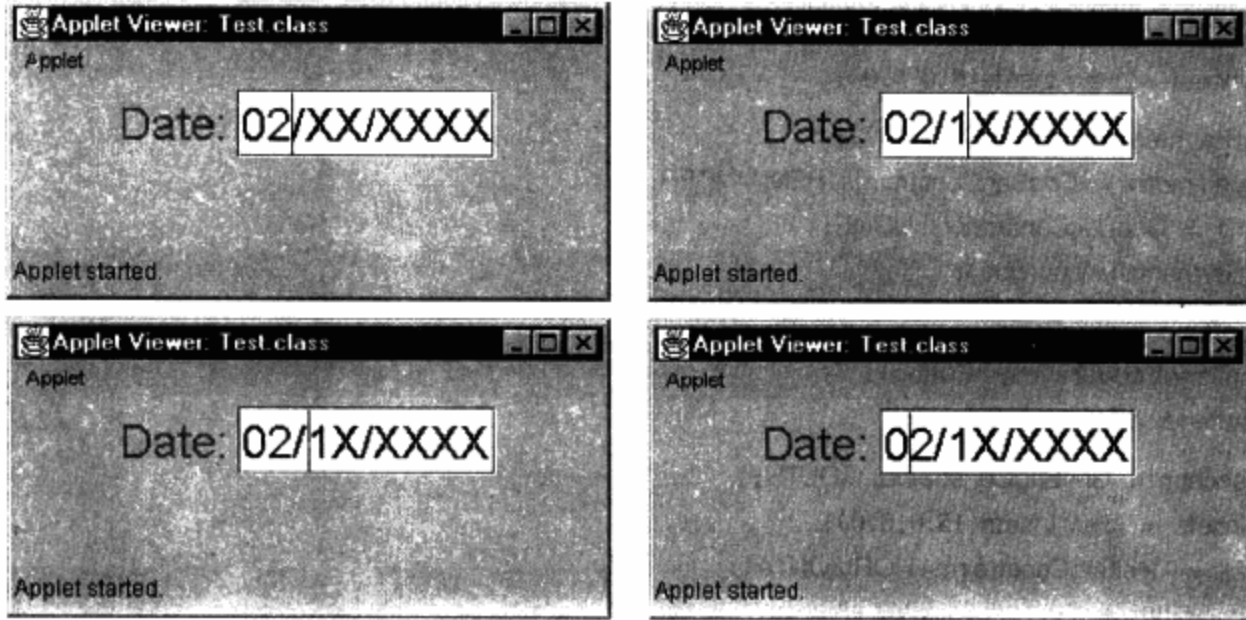


图 22-4 一个用于日期单行文本域的定制文档

图 22-4 示出的小应用程序用在 `DateDocument` 中定义的一个公共常量字符串来实例化一个单行文本域,并把这个单行文本域的文档设置为 `DateDocument` 的一个实例。

```
public class Test extends JApplet {
    JTextField tf = new JTextField( DateDocument.initString );
    public Test() {
        ...
        tf.setDocument( new DateDocument( tf ) );
        ...
    }
}
```

`DateDocument` 类重载 `insertString()` 和 `remove()`。`insertString` 方法进行检查以确保插入的字符是一个有效的整数,并在需要时调整加字符的位置。方法 `remove` 并不删除这个文档中的任何字符,只调整加字符的位置。

```
class DateDocument extends PlainDocument {
    public static String initString = "XX/XX/XXXX"; //Y10K!
    private static int sep1 = 2, sep2 = 5;
    private JTextComponent textComponent;
    private int newOffset;

    public DateDocument (JTextComponent tc) {
        textComponent = tc;
        try {
            insertString(0, initString, null);
        }
        catch (Exception ex) {}
    }

    public void insertString (int offset, String s,
```



```

        AttributeSet attributeSet)
        throws BadLocationException {
    if (s.equals(initString)) {
        super.insertString(offset, s, attributeSet);
    }
    else {
        try {
            Integer.parseInt(s);
        }
        catch (Exception ex) {
            return ; //only allow integer values
        }

        newOffset = offset;
        if (atSeparator(offset)) {
            newOffset + + ;
            textComponent.setCaretPosition(newOffset);
        }

        super.remove(newOffset, 1);
        super.insertString(newOffset, s , attributeSet);
    }
}

public void remove(int offset, int length)
        throws BadLocationException {
    if (atSeparator(offset))
        textComponent.setCaretPosition(offset-1);
    else
        textComponent.setCaretPosition(offset);
}

private boolean atSeparator(int offset) {
    return offset == sep1 || offset == sep2;
}
}

```

注意,一个更强壮 DateDocument 应该检查月、日和年项的有效性。例 22-4 列出了图 22-4 所示的小应用程序的完整代码。

例 22-4 实现一个定制文档

```

import javax.swing.* ;
import javax.swing.text.* ;
import java.awt.* ;
import java.awt.event.* ;

public class Test extends JApplet {
    JTextField tf = new JTextField(DateDocument.initString);

    public Test() {
        Container contentPane = getContentPane();
        Font font = new Font("Dialog", Font.PLAIN, 24);
        JLabel label = new JLabel("Date");

        tf.setDocument( new DateDocument(tf));

        label.setFont( font);
        tf.setFont( font);

        contentPane.setLayout( new FlowLayout());
    }
}

```

```

        contentPane.add( label );
        contentPane.add( tf );
    }
}

class DateDocument extends PlainDocument (
    public static String initString = "XX/XX/XXXX"; //Y10K!
    private static int sep1 = 2, sep2 = 5;
    private JTextComponent textComponent;
    private int newOffset;

    public DateDocument (JTextComponent tc) {
        textComponent = tc;
        try {
            insertString(0, initString, null);
        }
        catch (Exception ex) {}
    }

    public void insertString(int offset, String s,
        AttributeSet attributeSet)
        throws BadLocationException {
        if (s.equals(initString)) {
            super.insertString(offset, s, attributeSet);
        }
        else {
            try {
                Integer.parseInt(s);
            }
            catch (Exception ex) {
                return ; //only allow integer values
            }

            newOffset = offset;
            if (atSeparator(offset)) {
                newOffset ++ ;
                textComponent.setCaretPosition(newOffset);
            }
            super.remove(newOffset, 1);
            super.insertString(newOffset, s , attributeSet);
        }
    }

    public void remove(int offset, int length)
        throws BadLocationException {
        if (atSeparator(offset))
            textComponent.setCaretPosition(offset-1);
        else
            textComponent.setCaretPosition(offset);
    }

    private boolean atSeparator(int offset) {
        return offset == sep1 || offset == sep2;
    }
}

```

22.1.4 JTextField 组件总结

组件总结 22-1 总结了类 JTextField。

组件总结 22-1 **JTextField**

模型： java.swing.text.PlainDocument
UI 代表： javax.swing.plaf.basic.TextFiledUI
绘制器： _____
编辑器： _____
激发的事件： ActionEvents
替代： java.awt.TextFeild
类图：

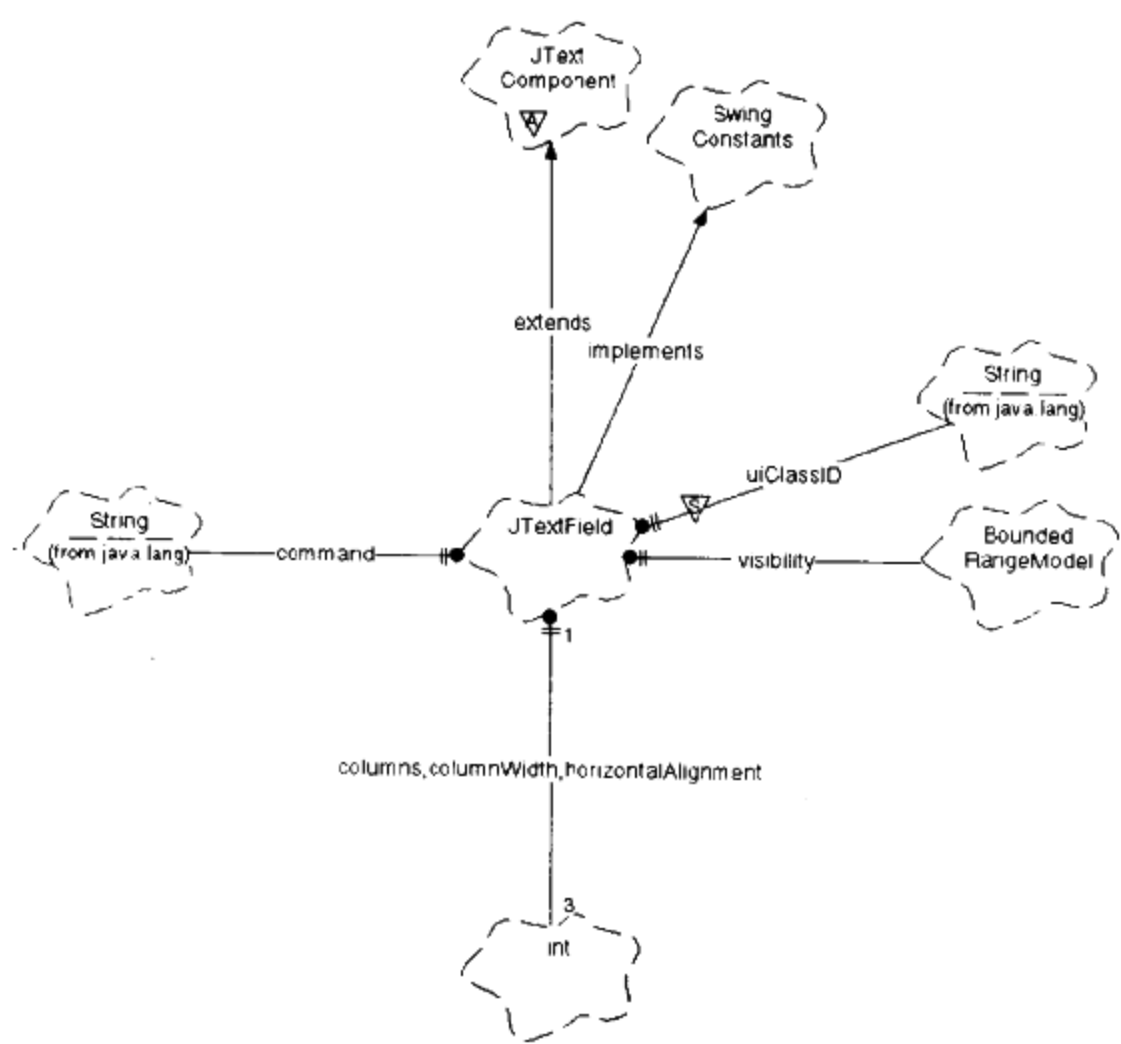


图 22-5 JTextField 的类图

JTextField 类扩展 JTextComponent 并实现 SwingConstants 接口。JTextField 维护对它的动作命令字符串和可视性(以 BoundedRangeModel 的形式)的 private 引用。并保留三个 integer 值,它们代表列数、列宽和文本的水平排列方式。

22.1.5 JTextField 属性

表 22-1 列出了由类 JTextField 维持的属性。

表 22-1 JTextField 属性

属性名	数据类型	属性类型 ^①	访问 ^②	默认值 ^③
actionCommand	String	S	SG	text field text
columns	int	S	SG	0
horizontalAlignment	int	B	SG	LEFT
horizontalVisibility	Bounded RangeModel	S	G	BoundedRangeModel

(续)

属性名	数据类型	属性类型 ^①	访问 ^②	默认值 ^③
scrollOffset	int	S	SG	0

- ① B = 关联的(激发 PropertyChangeEvent)/C = 受约束的/I = 索引的/
S = 简单的/Ch = 激发 ChangeEvent
- ② C = 可在创建时设置/G = 获取方法/S = 设置方法
- ③ L&F = 与界面样式有关

actionCommand ——当单行文本域激发一个动作事件时, **actionCommand** 被设置为这个事件的动作命令。如果没有显式地用 `JTextFied.setActionCommand` 方法来设置这个属性, 则 **actionCommand** 的缺省值为这个单行文本域的内容。

columns ——列宽被设置为单行文本域中当前字体的字符 **m** 的宽度。因此, 单行文本域的列宽是一个近似值。

horizontalAlignment ——确定在单行文本域内显示的文本的水平排列方式。有效值是:

- `JTextField.LEFT`
- `JTextField.CENTER`
- `JTextField.RIGHT`

文本的竖直排列方式是居中显示, 不能设置。

horizontalVisibility ——确定单行文本域的哪个部分是可视的。通常, 不直接访问 **horizontalVisibility** 属性, 参见 **scollOffset** 属性。

scrollOffset ——确定单行文本域的哪一部分是可见的。滚动偏移由单行文本域的水平可视性作为可视值来维护, 参见 **horizontalVisibility** 属性。

22.1.6 **JTextField 事件**

缺省时, 当单行文本域有焦点并按下回车键后将激发一个动作事件。如果单行文本域在一

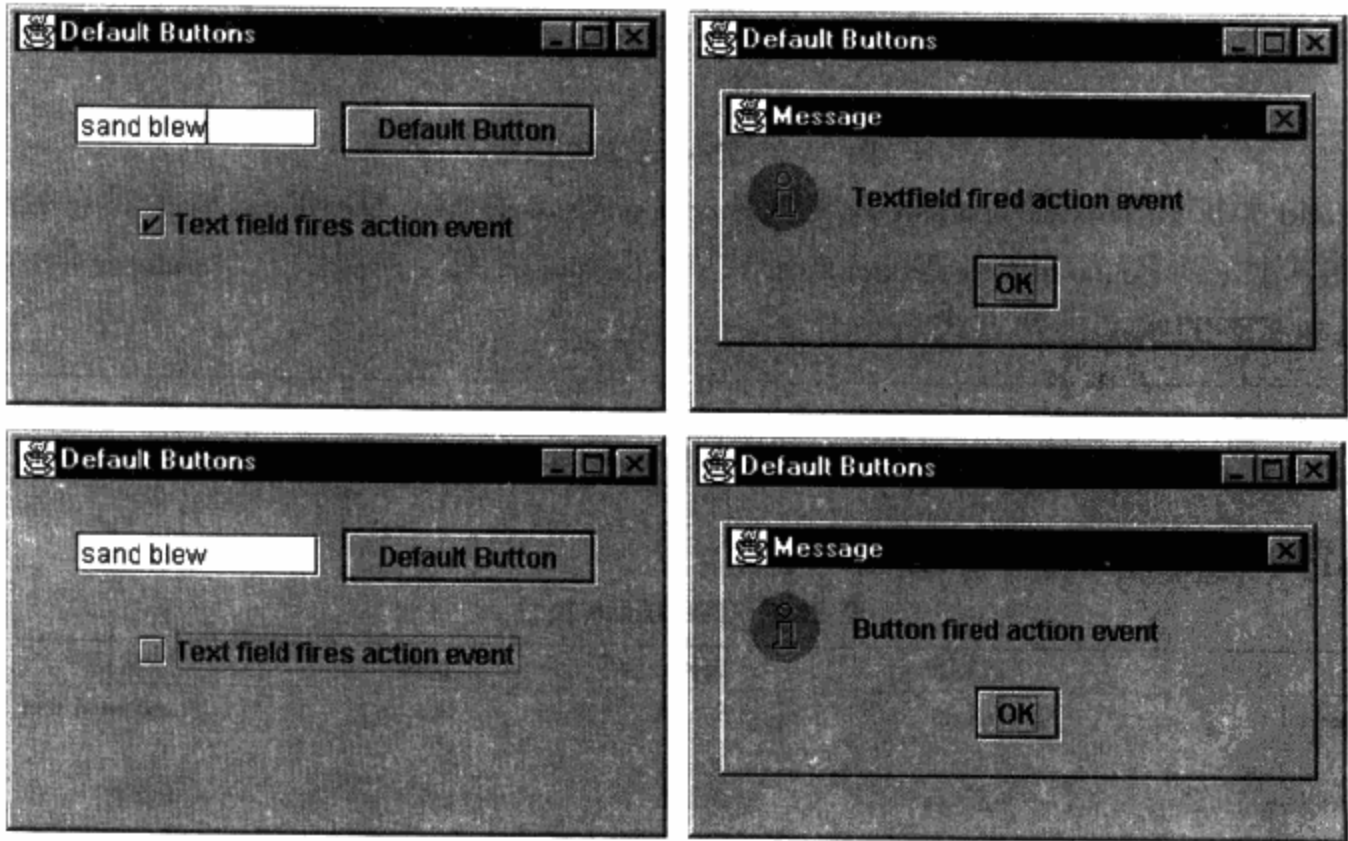


图 22-6 单行文本域和缺省按钮

个有缺省按钮的窗口里,则在这个单行文本域里按下回车键将激发这个单行文本域的动作事件,而不是激活这个按钮。幸运的是,修改单行文本域的键映射很简单,这样它就不会激发这个动作事件。

图 22-6 所示的应用程序提供了一个复选框,这个复选框根据它的选取状态来添加或删除 Enter 键的键击捆绑。如果选取了这个复选框,则 Enter 键击捆绑在这个单行文本域的键映射中,此时在这个单行文本域中按下 Enter 键将使这个单行文本域激发一个动作事件。如果这个复选框没有被选中,则在这个单行文本域中按下 Enter 键将由缺省按钮来处理,这个按钮也激发一个动作事件。

图 22-6 的上图显示了这个单行文本域激发一个动作事件的情况,下图显示了缺省按钮激发这个事件情况。

例 22-5 列出了图 22-6 所示应用程序的完整代码。

例 22-5 单行文本域和缺省按钮

```
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JFrame {
    private JTextField field = new JTextField(10);
    private JButton b = new JButton("Default Button");
    private JCheckBox cb = new JCheckBox(
        "Text field fires action event");

    public Test() {
        Container contentPane = getContentPane();

        SwingUtilities.getRootPane(this).setDefaultButton(b);
        cb.setSelected(true);

        contentPane.setLayout( new FlowLayout(
            FlowLayout.CENTER, 10, 20));

        contentPane.add ( field)
        contentPane.add( b);
        contentPane.add( cb);

        b.addActionListener ( new ActionListener() {
            public void actionPerformed( ActionEvent e) {
                JOptionPane.showMessageDialog( Test.this,
                    "Button fired action event");
            }
        });

        field.addActionListener ( new ActionListener() {
            public void actionPerformed( ActionEvent e) {
                JOptionPane.showMessageDialog( Test.this,
                    "Textfield fired action event");
            }
        });

        cb.addActionListener ( new ActionListener() {
            private Keymap km;
            private KeyStroke ks;
            private Action action;
```

```

    public void actionPerformed( ActionEvent e) {
        if (cb.isSelected()) {
            km.addActionForKeyStroke(ks, action);
        }
        else {
            if (ks == null) {
                km = field.getKeymap();
                ks = KeyStroke.getKeyStroke(
                    KeyEvent.VK_ENTER, 0);
                action = km.getAction(ks);
            }
            km.removeKeyStrokeBinding(ks);
        }
    }
};

public static void main(String args[]) {
    GJApp.launch ( new Test(),
        "Default Buttons", 300, 300, 350, 200);
}

```

这个应用程序创建一个单行文本域、一个按钮和一个复选框,并把它依次添加到这个应用程序的内容面板中。这个应用程序用 `JRootPane.setDefaultButton` 方法把这个按钮指定为缺省按钮。通过调用 `SwingUtilities.getRootPane()` 来获得这个根面板。有关 `SwingUtilities` 类的更多信息,请参见 6.3 节“Swing 的实用工具”。

这个按钮和这个单行文本域都配备动作监听器,这些监听器显示一个消息对话框来指示这个动作的源。这个复选框有一个动作监听器,这个动作监听器在复选框没有被选中时删除 Enter 键的键击捆绑,在选中复选框时恢复这个键击。

22.1.7 JTextField 类总结

类总结 22-1 总结了 `JTextField` 类

类总结 22-1 JTextField

扩展: `JTextComponent`

实现: `SwingConstants`

1. 常量

`public static final String notifyAction`

`notifyAction` 字符串常量代表当提交单行文本域文本的变化时激发的动作名字^①,可以用 `notifyAction` 字符串常量来获取对这个动作的一个引用。

2. 构造方法

`public JTextField()`

`public JTextField(int columns)`

`public JTextField(String initialString)`

`public JTextField(String initialString, int columns)`

① 缺省时,通过按 `KeyEvent.VK_ENTER` 键提交变化。

```
public JTextField( Document, String initialString, int columns )
```

在把一个单行文本域进行实例化时,可以指定单行文本域的初始字符串、列数和文档。无参数的构造方法构造一个列数为 0 的单行文本域,即当根据单行文本域的首选大小布局时,只有单行文本域的边框是可见的。

3. 方法

(1) 动作事件

```
public synchronized void addActionListener( ActionListener )
public synchronized void removeActionListener( ActionListener )
protected void fireActionPerformed()
public void postActionEvent()
```

与其他的激发动作事件的 Swing 组件一样,JTextField 提供了一些登记动作监听器和把动作事件发送给已登记的监听器的方法。PostActionEvent 方法是一个 public 方法,它只调用保护方法 fireActionPerformed。postActionEvent 由单行文本域的 UI 代表调用。

(2) 缺省模型/动作/动作命令

```
protected Document createDefaultMode()
public Action[ ] getAvtion()
public void setActionCommand( String )
```

方法 createDefaultMode 创建 PlainDocument 的一个实例。JTextField 的扩展可以重载 createDefaultMode 来安装定制模型。

JTextField 重载 JtextComponent 的 getAction() 来把单行文本域的动作添加到由 JTextComponent 提供的动作集中。如例所示,JTextField 使用 TextAction.augmentList 方法:

```
// from JTextField.java:
public Action[ ] getActions() {
    return TextAction.augmentList( super.getActions(), defaultActions );
}
```

有关文本动作的更多信息,请参见 21.2 节“动作”,例 23-4 给出了在编辑器工具包中使用 TextAction.augmentList 方法的例子。

单行文本域的动作命令是一个嵌入在被激发的动作事件中的字符串(当提交这个单行文本域的变化时)。当激发这个动作事件时,这个动作命令缺省设置在单行文本域中显示的文本。可以用 setActionCommand 方法来显式地设置这个动作命令。

(3) 列/字体

```
public int getColumns()
public void setColumns( int width )
protected int getColumnWidth()
public void setFont( Font )
```

单行文本域的列数可以在构造时或在构造后被显式地设置。列被定义为单行文本域字体的字符 m 的宽度。对有不同字体宽度的单行文本域而言,显式地设置列数是一种近似值。

可以通过方法 getColumnWidth 来获得列宽(以像素为单位)。重载 setFont 方法以重新设置这个单行文本域的列宽,以便在下一次调用 getColumnWidth 时,将重新计算列宽。

(4) 排列方式/可视性/滚动偏移

```
public int getHorizontalAlignment()
public void setHorizontalAlignment( int alignment )
public BoundedRangeModel getHorizontalVisibility()
```



```
public int getScrollOffset()
public void setScrollOffset( int offset )
```

可以用下面所列的常量之一来指定单行文本域的水平排列方式:

- JTextField.LEFT
- JTextField.CENTER
- JTextField.RIGHT

调用 setHorizontalAlignment 将使单行文本域无效并进行重绘。

单行文本域的水平可视性由 BoundedRangeModel 的一个实例来构造,这个实例通过方法 getHorizontalAlignment 来访问。模型的值是滚动偏移量,它是单行文本域左上角显示的文本的位置。滚动偏移可以用上面所列的最后两个方法来访问。

(5)可访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext()
public String getUIClassID()
```

与其他 Swing 组件一样,JTextField 提供了用 getAccessibleContext 方法来访问它的可访问相关内容。与其他轻量 Swing 组件一样,JTextField 实现了 getUIClassID(),它返回一个字符串——TextField,这个字符串标识单行文本域 UI 代表的类。

(6)其他方法

```
public Dimension getPreferredSize()
public boolean isValidRoot()
public void scrollRectToVisible( Rectangle )
protected String paramString()
```

JTextField 重载方法 getPreferredSize 来考虑是否已显式地设置了列数。首选高度被设置为由 super.getPreferredSize() 返回的高度。如果没有显式地设置列数,还可以通过调用 super.getPreferredSize() 来获得首选宽度,如果这样,首选宽度是列数乘以列宽。

JTextField 重载方法 isValidRoot 以返回 true。这就是说,为单行文本域调用 revalidate() 方法将使这个域重新有效但却不能使这个域(在滚动窗格或根窗格中)的兄弟和父重新有效。有关 revalidate 方法的更多信息,请参见 4.3.5 的“Validate、Invalidate 和 Realidate 方法”一节。

scrollRectToVisible 方法使用 horizontalVisibility 关联范围属性来滚动单行文本域中的文本。paramString 方法返回一个代表单行文本域的字符串。

Swing 提示

单行文本域和缺省按钮

在 Swing 中,当与缺省按钮在同一个窗体中的一个组件有焦点时,用户按下回车键将激活缺省按钮。缺省时,单行文本域在它拥有焦点并按下回车键时将激发动作事件,因此,鼠标按下事件不激活缺省按钮。

为了在单行文本域有焦点并按下回车键时激活缺省按钮,可以从单行文本域的键映射中删除回车键的键击捆绑。例如:

```
km = textfield.getKeymap();
KeyStroke ks = km.getKeyStroke( KeyEvent.VK_ENTER, 0 );
km.removeKeyStrokeBinding(ks);
```

22.1.8 AWT 兼容

JTextField 和 java.awt.TextField 在源代码上几乎是兼容的,如表 22-2 所示。只有 AWT 单行文本域的回显字符方法没有在 JTextField 中得到实现。JTextField 不支持隐藏输入,有关口令域的更多信息请参见“JPasswordField”。

表 22-2 java.awt.TextField 方法和 JTextField 的等价方法

java.awt.TextField 方法	JTextField 的等价方法
void addActionListener(ActionListener)	void addActionListener(ActionListener)
boolean echoCharsSet()	参见 JPasswordField
int getColumns()	int getColumns()
char getEchoChar()	参见 JPasswordField
void remove ActionListener(ActionListener)	void remove Action(Action Listener)
void setColumns(int)	void setColumns(int)
setEchoChar()	参见 JPasswordField
void setText(String)	void setText(String)

22.2 JPasswordField

Swing 的口令域把输入到这个域的字符隐藏起来并显示“*”。可以在创建口令域之后设置星号为回显字符。

口令域的使用很简单。如图 22-7 所示的小应用程序所说明的。这个小应用程序包含一个标签和一个口令域,这个口令域的回显字符被设置为“?”。把一个动作监听器添加到这个域中,它根据所提交的口令来更新这个小应用程序的状态条。

例 22-6 列出了图 22-7 所示小应用程序的完整代码。

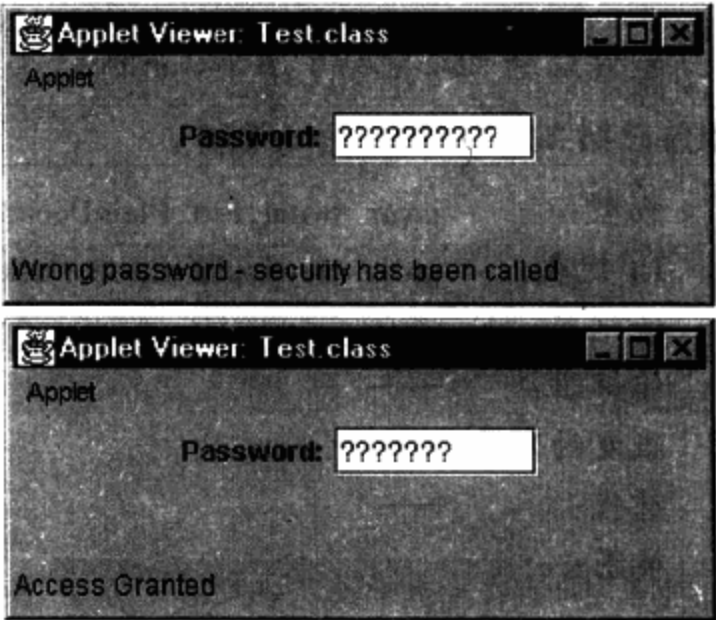


图 22-7 使用 JPasswordField

例 22-6 使用 JPasswordField

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
import java.util.* ;

public class Test extends JApplet {
    private String pw = "do142ce";
    private JPasswordField passwordField = new JPasswordField(8);

    public void init() {
        Container contentPane = getContentPane();
        JPanel panel = new JPanel();

        panel.add( new JLabel("Password:")) ;
        panel.add( passwordField );

        passwordField.setEchoChar('?');

        contentPane.add(panel, BorderLayout.CENTER);

        passwordField.addActionListener( new ActionListener() {
            public void actionPerformed ( ActionEvent e ) {
```

```
String password = new String (
    passwordField.getPassword());
if (pw.equals(password))
    ShowStatus("Access Granted");
else
    ShowStatus ("Wrong password-security " +
        "has been called");
    }
};
```

这个小应用程序构造一个列宽为 8 的 JPasswordField 的实例,一个动作监听器从口令域中提取口令并检查这个口令是否有效。

22.2.1 JPasswordField 组件总结

组件总结 22-2 总结了 JPasswordField 类。

组件总结 22-2 JPasswordField

- 模型: javax.swing.text.PlainDocument
- UI 代表: javax.swing.plaf.basic.BasicPasswordFieldUI
- 绘制器: ——
- 编辑器: ——
- 激发的事件: ActionEvents
- 替代: ——
- 类图:

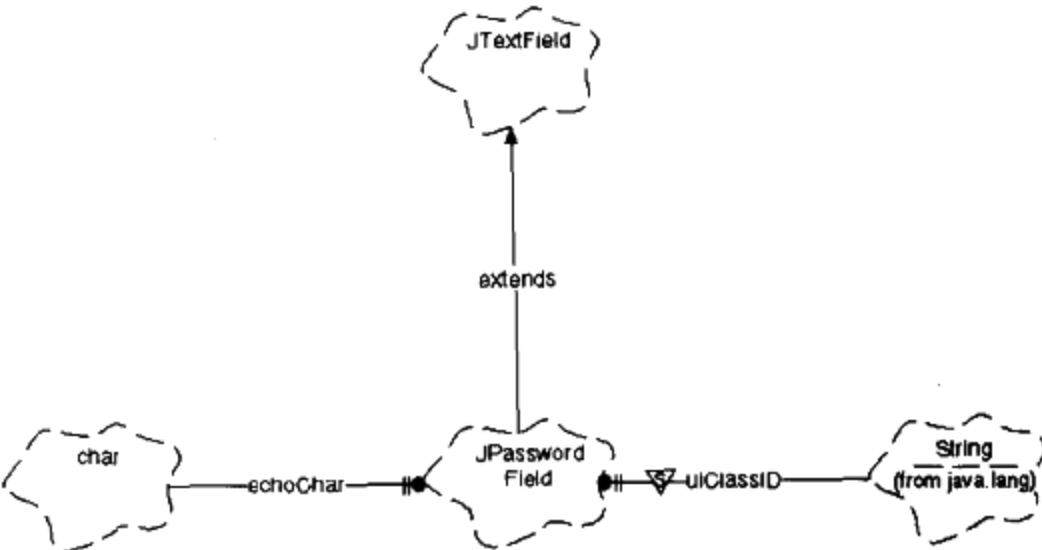


图 22-8 JTextField 的类图

JPasswordField 是 JTextField 的一个简单扩展,它维护一个回显字符。与它的超类一样,JPasswordField 有一个 PlainDocument 实例来作为它的缺省文档。

22.2.2 JPasswordField 属性

表 22-3 列出了由类 JPasswordField 维护的属性。

表 22-3 JPasswordField 的属性

属性名称	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
echoChar	char	S	SG	L&F
password	char[]	S	G	-

- ① B = 关联的(激发 PropertyChangeEvent)/C = 受约束的/I = 索引的/S = 简单的/Ch = 激发 ChangeEvent
② C = 可在创建时设置/G = 获取方法/S = 设置方法
③ L&F = 与界面样式有关

echoChar —— 当输入一个字符时在口令域中显示的字符。
password —— 一个字符数组,它包含那些输入到口令域中的字符。

22.2.3 JPasswordField 类总结

类总结表 22-2 总结了类 JpasswordField。

类总结表 22-2 JPasswordField

扩展:JTextFeild

1. 构造方法

```
public JPasswordField()  
public JPasswordField( int columns )  
public JPasswordField( String initialString )  
public JPasswordField( String initialString , int columns )  
public JPasswordField( Document, String initialString , int columns )
```

JPasswordField 提供与它超类 JtextField 相同的构造方法集。JPasswordField 的构造方法调用超类的构造方法并把回显字符设置为“*”。

2. 方法

(1)口令/安全

```
public char[ ] getPassword()  
public void copy()  
public void cut()  
public String getText()  
public String getText( int p0, int p1 ) throws BadLocationException
```

应该使用方法 getPassword 获取口令而不是使用自 JtextField 继承的 getText 方法。如果使用 getText,在编译时将得到一个警告。

重载 copy 和 cut 方法以简单地发声,以便不能程序地操纵口令。

(2)回显字符

```
public char getEchoChar()  
public void setEchoChar()  
public boolean echoCharIsSet()
```

这三个方法是口令域的回显字符的访问方法。这些方法与 java.awt.TextField 的方法有相同的原型,以便维护与 AWT 的兼容性。

(3)可访问性/插入式界面样式

```
public AccssibleContext getAccessibleContext()  
public String getUIClassID()  
protected String paramString()
```

与所有其他的 Swing 组件一样,JPasswordField 提供了访问它的相关内容的方法 getAccessibleContext。与所有其他的轻量 Swing 组件一样,JPasswordField 实现了 getUIClassID,它返回一个字符

串——PasswordField, 这个字符串标识口令域 UI 代表的类。
paramString 方法返回口令域的文本描述。

22.3 JTextArea

Swing 的多行文本域(由类 JTextArea 表示)可以显示多行纯文本,多行文本域被定义为具有一种字体和一种颜色的文本。多行文本域可以在字符或字边界处换行。JTextArea 类与 java.awt.TextArea 类在源代码上几乎是兼容的。

JTextArea 是一个简单的组件,它没有提供任何复杂的功能,这些功能在 Swing 的其他多行文本组件中可以找到,这些组件如 JEditorPane 和 JTestPane。但是,在很多情况下它仍然是很有用的,如图 22-9 所示的应用程序所说明的那样。

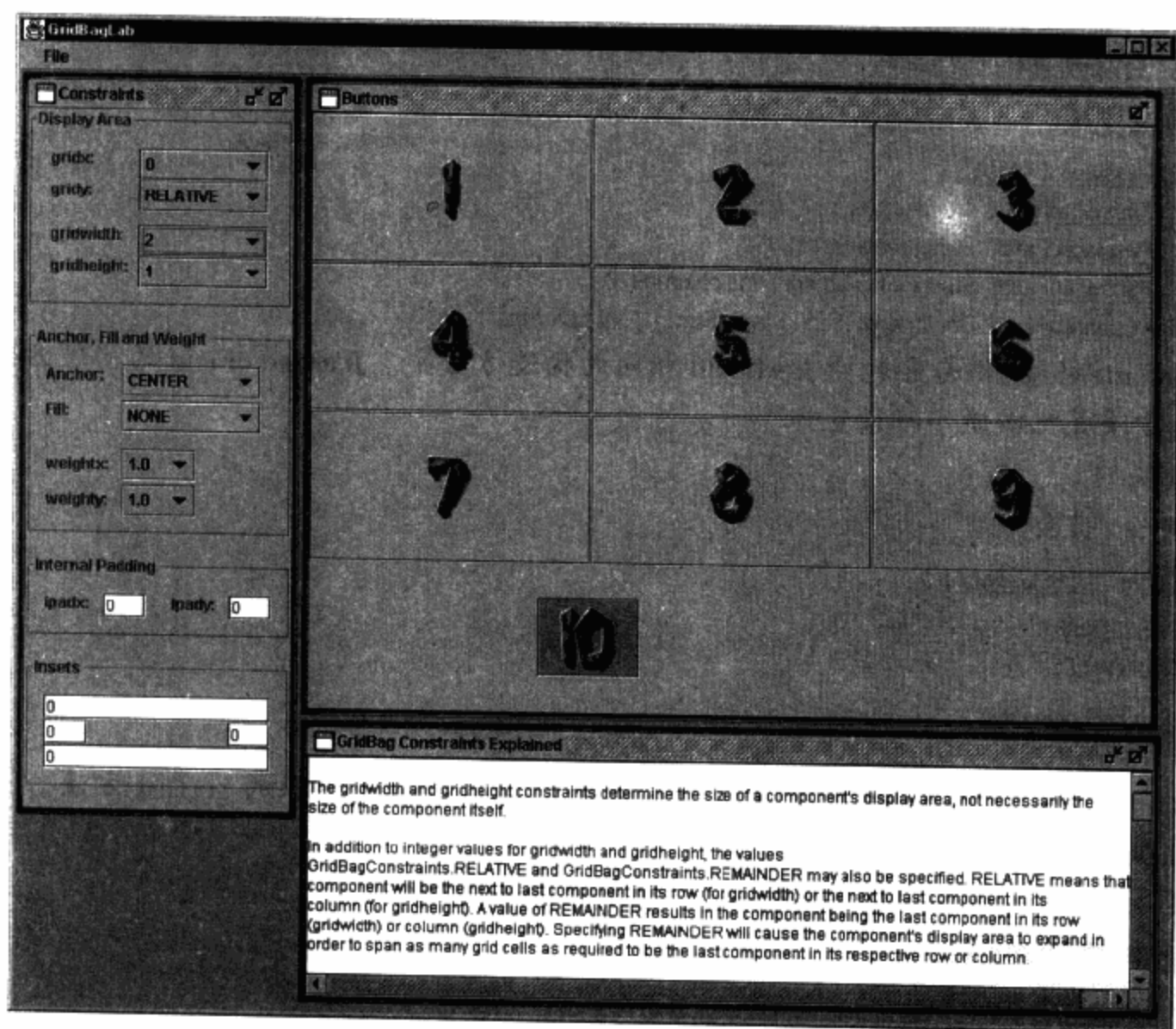


图 22-9 GridBagLab 使用一个多行文本域

图 22-9 所示的应用程序是一个类别实验室,用于探索 GridBagLayout 布局管理器。这个应用程序使用一个在滚动窗格中的多行文本域来显示有关格子约束条件的信息(当加字符在 Constraints 窗口的项上移动时)。

这个应用程序用一个文件的内容来创建一个 JTextArea 实例。通过调用 JTextArea.setWrapStyleWord(true)使这个多行文本域在字边界处能换行。设置了这个多行文本域的字体,而且把这个多行文本域的可编辑性设置为 false。

与其他的 Swing 组件一样,JTextArea 没有实现自己的滚动方法,但它实现了一个 Scrollable 接口。在需要滚动时,应该把这个多行文本域放在一个滚动窗格中。

```
public class GridBagLab extends JFrame {
    ...
    static JTextArea helpTextArea =
        new JTextArea(contentsOfFile("intro.txt"));
    ...
    helpTextArea.setWrapStyleWord(true);
    helpTextArea.setEditable(false);
    helpTextArea.setFont(
        new Font("Times-Roman", Font.PLAIN, 12));
    ...
    helpFrame.getContentPane().add(
        new JScrollPane(helpTextArea), "Center");
}
...
```

图 22-10 所示的小应用程序说明了多行文本域的换行情况。这个小应用程序包括一个多行文本域并提供了三个单选按钮来控制这个多行文本域的换行行为。图 22-10 的左上图显示了这个小应用程序初始的样子,右上图和下图分别显示了这个多行文本域在字符和字处换行的效果。

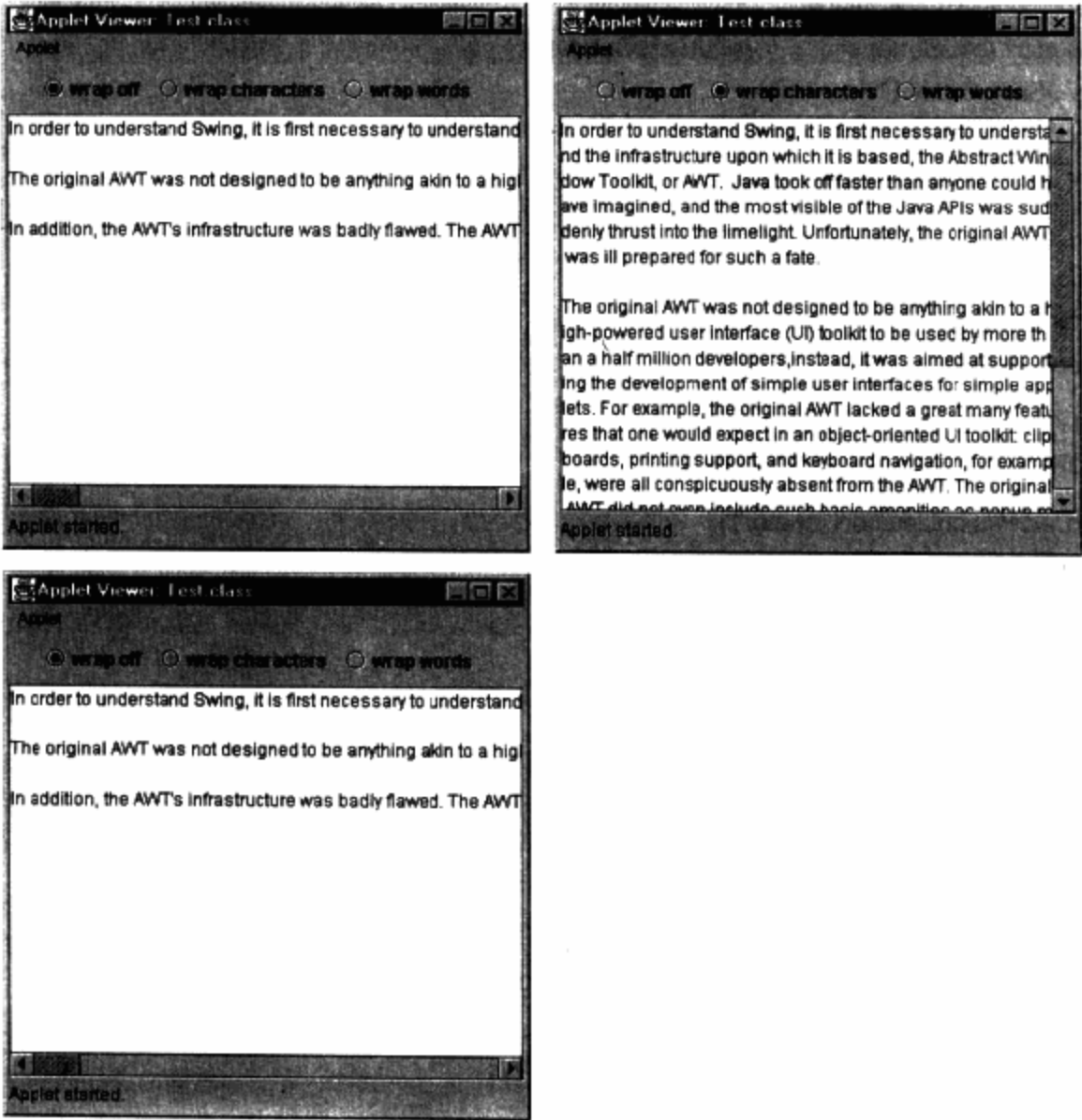


图 22-10 多行文本域的换行

例 22-7 列出了图 22-10 所示小应用程序的代码。

例 22-7 指定 JTextArea 的换行行为

```
import java.awt.*;
```

```

import java.awt.event.*;
import javax.swing.*;
import java.swing.text.*;
import java.io.FileReader;

public class Test extends JApplet {
    private JTextArea textArea = new JTextArea();
    private Container contentPane = getContentPane();
    public void init() {
        To(textArea, "text");
        contentPane.add(new ControlPanel(), BorderLayout.NORTH);

        contentPane.add(new JScrollPane(textArea),
            BorderLayout.CENTER);
    }
    private void readFile(JTextComponent textcomponent, String s) {
        try { ( new DefaultEditorKit().read(
            new FileReader(s), textComponent.getDocument(), 0 );
        } catch ( Exception ex ) { ex.printStackTrace(); }
    }
    class ControlPanel extends JPanel {
        JRadioButton radioButtons[] = new JRadioButton[] {
            new JRadioButton("wrap off"),
            new JRadioButton("wrap characters"),
            new JRadioButton("wrap words"),
        };
        public ControlPanel() {
            ButtonGroup group = new ButtonGroup();
            Listener listener = new Listener();

            for (int i = 0; i < radioButtons.length; ++i) {
                JRadioButton b = radioButtons[i];

                b.addActionListener(listener);
                group.add(b);
                add(b);

                if ( i == 0 )
                    b.setSelected(true); // "wrap off"
            }
        }
        class Listener implements ActionListener {
            public void actionPerformed ( ActionEvent e ) {
                String action = e.getActionCommand();

                textArea.setLineWrap(! action.equals("wrap off"));
                textArea.setWrapStyleWord(
                    action.equals("wrap words"));
            }
        }
    }
}

```

这个小应用程序把一个动作监听器添加到每一个单选按钮中,这些监听器启用或禁止换行并所选中的按钮来设置换行风格。

22.3.1 JTextArea 组件总结

组件总结 22-3 总结了类 JTextArea。

组件总结 22-3 JTextArea

模型: javax.swing.text.PlainDocument
UI 代表: javax.swing.plaf.basic.BasicTextAreaUI
绘制器: ——
编辑器: ——
激发的事件: ——
替代: java.awt.TextArea
类图:

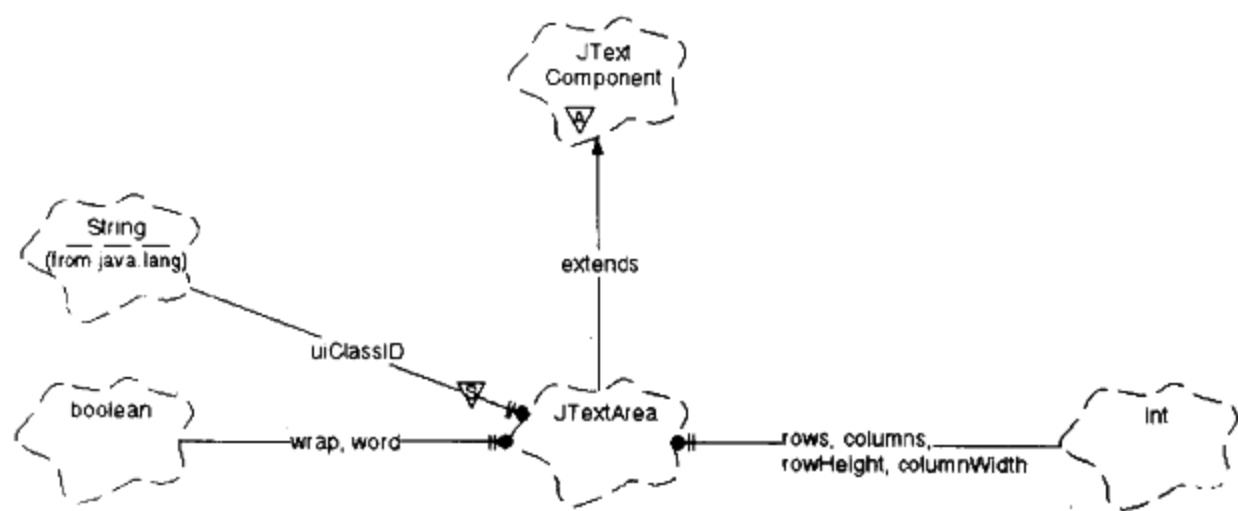


图 22-11 JTextArea 的类图

类 JTextArea 扩展 JTextComponent 并维护 private boolean 值,这些值维护多行文本域的换行状态。JTextArea 还维护代表行数、列数、行高和列宽的 integer 值。

22.3.2 JTextArea 属性

表 22-4 示出了由 JTextArea 类维护的属性。

表 22-4 JTextArea 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
columns	int	S	CSG	0
lineCount	int	S	G	0
lineWrap	boolean	B	SG	false
rows	int	S	SG	0
tabSize	int	B	SG	from document
wrapStyleWord	boolean	B	SG	false

① B = 关联的(激发 PropertyChangeEvent)/C = 受约束的/I = 索引的/
S = 简单的/Ch = 激发 ChangeEvent
② C = 可在创建时设置/G = 获取方法/S = 设置方法
③ I&F = 与界面样式有关

columns——多行文本域的显示列数。与单行文本域一样,列宽度是多行文本域的当前字体

的 m 字符的宽度。

lineCount——多行文本域中包含的文本行数。

lineWrap——决定文本是否能在多行文本域的右边界处换行。换行类型由 wrapStyleWord 属性来确定。

rows——在多行文本域中显示的文本行数。

tabSize——按下 Tab 键时所能插入的字符数。

wrapStyleWord——一个布尔属性。当其值为 true 时,以单词为界进行换行;当其值为 false 时,以字符为界进行换行。

22.3.3 JTextArea 类总结

类总结 22-3 类总结了 JTextArea 类。

类总结表 22-3 JTextArea

扩展: JTextComponent

1. 构造方法

```
public JTextArea()
public JTextArea( int rows, int columns )
public JTextArea( String initialString )
public JTextArea( String initialString, int rows, int columns )
public JTextArea( Document )
public JTextArea( Document, String initialString, int rows, int columns )
```

与 JTextField 一样, JTextArea 提供了许多构造方法来指定文档、初始字符串和列数。JTextArea 还允许在构造时设置文本的行数。

无参数的构造方法构造一个 0 行 0 列的多行文本域,因此,如果把它的尺寸设置为首选大小,则它小得几乎看不见。

2. 方法

(1) 文档/文本/Tab 大小

```
protected Document createDefaultModel()
public void append( String )
public void insert( String int start )
public void replaceRange( String, int start, int end )
public int getTabSize()
public void setTabSize( int )
```

与 JTextField 一样,多行文本域的缺省模型是 PlainDocument 的一个实例。上面所列的第二组方法可以在不访问多行文本域的文档的情况下向多行文本域追加、插入和替换字符串。

Tab 大小代表当按下 Tab 键时所插入的字符数,可以用最后一组方法来访问 Tab 的大小。

(2) 列数和行数

```
protected int getColumnWidth()
public int getColumns()
public void setColumns( int )
protected int getRowHeight()
public int getRows()
public void setRows( int )
```

可以用上面所列的方法来访问以像素为单位的行数、列数以及行高和列宽。如果没有显式地设置行数和列数,那么 getColumns() 和 getRow() 分别返回 0。行高由多行文本域字体 metrics 来

指定,列宽是多行文本域当前字体中 m 字符的宽度。

(3) 行/换行

```
public int getLineCount()

public int getLineEndOffset( int line ) throws BadLocationException
public int getLineStartOffset( int line ) throws BadLocationException
public int getLineOfOffset( int offset ) throws BadLocationException

public boolean getLineWrap()
public void setLineWrap( boolean )

public boolean getWrapStyleWord()
public void setWrapStyleWord( boolean )
```

由 `getLineCount()` 返回的行数代表多行文本域中的行数。

上面所列的第二组方法可以用来获得一个行号,给定一个偏移量和给定行的开始和结束偏移量,偏移量被转换为模型中的位置。

上面所列的最后一组方法访问 `lineWrap` 和 `wrapStyleWord` 属性,`lineWrap` 和 `wrapStyleWord` 属性在 22.3.2 节“`JTextArea` 属性”中进行介绍。

(4) Scrollable 接口

```
public Dimension getPreferredSize()
public boolean getScrollableTracksViewportWidth()
public int getScrollableUnitIncrement( Rectangle visibleRect, int orientation, int direction )
```

`getPreferredSize` 方法返回在滚动窗格中的多行文本域的视口首选大小。如果显式地设置了行数和列数,那么首选行高是行数乘以行高,首选列宽是列数乘以列宽。如果没有显式地设置行数和列数,那么首选视口大小从 `JTextArea` 的超类 (`JTextComponent`) 中获得。

`getScrollableTracksViewportWidth` 方法从 `JTextComponent` 重载而来,它考虑了换行。如果允许换行,则这个方法返回 `true`,指示多行文本域的宽度应该保持与多行文本域的视口的宽度相同。

`getScrollableUnitIncrement` 方法根据传送给它的滚动方向参数来返回列宽或行高。

(5) 其他方法

```
public Dimension getPreferredSize()
public boolean isManagingFocus()
protected void processComponentKeyEvent( KeyEvent )
public void setFont( Font )
protected String paramString()
```

与 `JTextField` 一样,`JTextArea` 重载 `getPreferredSize` 方法来考虑是显式地设置了行数还是显式地设置了列数。如果显式地设置了行数,则首选高度用行数乘以行高来计算。如果显式地设置了列数,则首选宽度用列数乘以单列宽来计算。

`JTextArea` 重载 `isManagingFocus()` 以便返回 `true`,即当按下 Tab 键时,焦点不会移到下一个有焦点的组件上。有关焦点管理器和 `isManagingFocus` 方法的详细内容,请参见 4.10 节“焦点管理”。重载 `processComponentKeyEvent` 以确保按下了 Tab 和 Shift + Tab 键。

与 `JTextField` 一样,`JTextArea` 重载 `setFont()` 以便在字体变化时能把行高和列宽重新设置为 0。这些值被重新设置以便接下来对 `getRowHeight` 和 `getColumnWidth` 的调用将重新计算这些值。

(6) 访问性/插入式界面样式

```
public AccessibleContext getAccessibleContext()
public String getUIClassID()
```

与所有其他的 Swing 组件一样, `JTextArea` 提供了 `getAccessibleContext` 方法, 这个方法可以访问其可访问性的相关内容。与所有其他的轻量 Swing 组件一样, `JTextArea` 实现 `getUIClassID()` 方法, 这个方法返回一个字符串——“`TextArea`”, 它标识多行文本域 UI 代表的类。

22.3.4 AWT 兼容

`JTextArea` 代码与 `java.awt.TextArea` 代码是兼容的(除两个没有被 `JTextArea` 实现的方法外), 如表 22 - 5 所示。因为 Swing 的多行文本域没有直接实现滚动, 所以 `java.awt.TextArea.getScrollbarVisibility()` 方法没有相应的 `JTextArea` 方法。

表 22-5 `java.awt.TextArea` 和 `JTextArea` 的对应方法

java.awt.TextArea 方法	JTextArea 的对应方法
<code>void append(String)</code>	<code>void append(String)</code>
<code>int getColumns()</code>	<code>int getColumns()</code>
<code>int getRows()</code>	<code>int getRows()</code>
<code>char getScrollbarVisibility()</code>	——
<code>int getScrollbarVisibility(int)</code>	——
<code>void insert(String, int)</code>	<code>void insert(String, int)</code>
<code>void replaceRange (String, int, int)</code>	<code>void replaceRange (String, int, int)</code>
<code>void setColumns(int)</code>	<code>void setColumns(int)</code>
<code>void setRows(int)</code>	<code>void setRows(int)</code>

22.4 JEditorPane

编辑器窗格不仅能够显示多行文本还能够显示不同类型的内容。缺省情况下, 编辑器窗格可以显示纯文本、HTML 和 RTF 文本。编辑器窗格是 `JEditorPane` 类的实例, `JEditorPane` 类是 `JTextComponent` 的一个直接扩展。

必须理解有关 `JEditorPane` 类的两个问题。第一, `JEditorPane` 显示 HTML 和 RTF 格式文本时并不特别好, 在以后的版本中将改进对 HTML 和 RTF 格式文本的支持, 但在 `Swing1.1FCS` 中, 对 HTML 和 RTF 格式文本的支持的实现是不成熟。第二, `JEditorPane` 是 `JTextPane` 的超类, 并且 `JEditorPane` 和 `JTextPane` 从根本上有别于 `JTextField` 和 `JTextArea`。有关编辑器窗格或文本窗格和单行文本域/多行文本域从根本上的更多信息, 请参见本章 Swing 提示“编辑器窗格和编辑器工具包”。

图 22-12 所示的应用程序使用一个编辑器窗格来显示一个 HTML 文件。

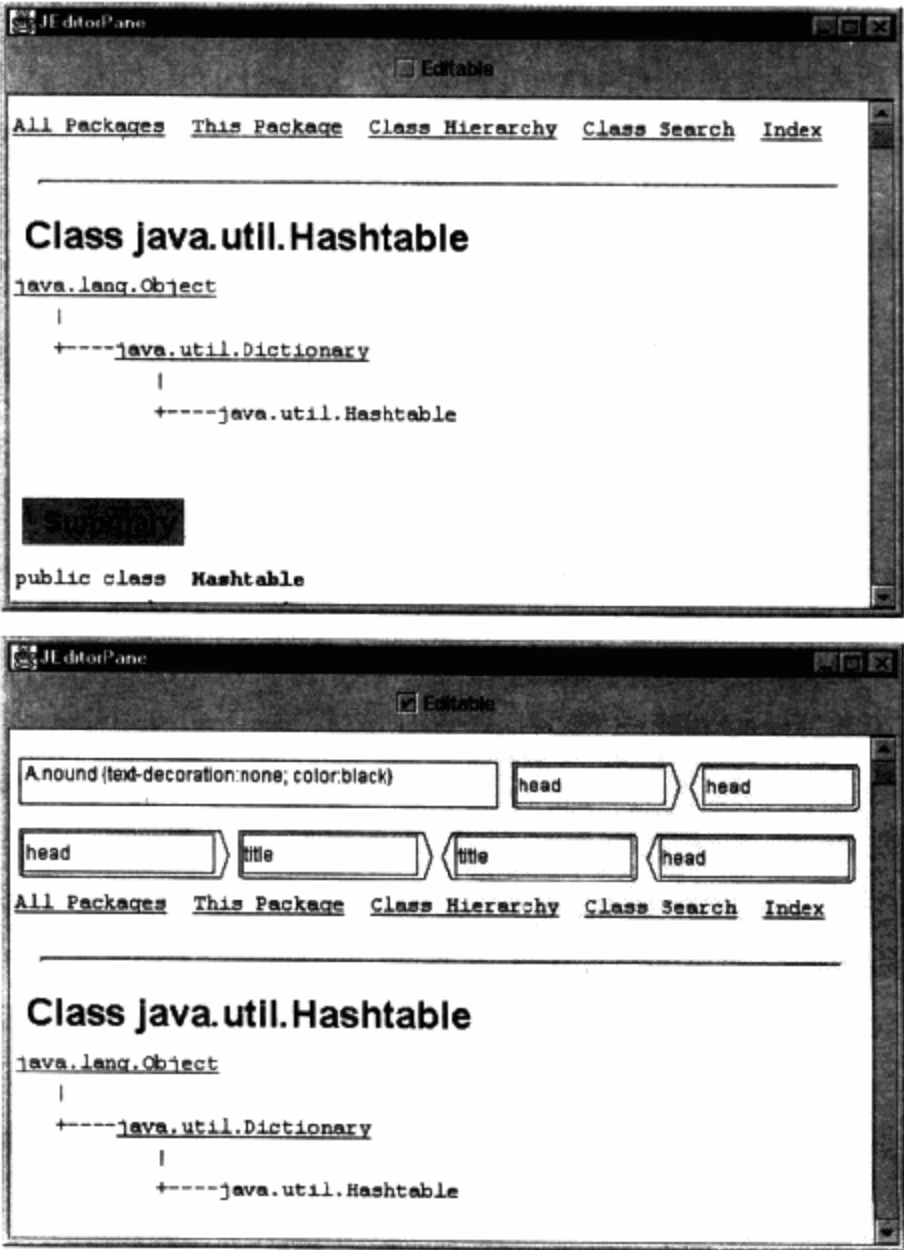


图 22-12 一个有 HTML 内容的编辑器窗格

根据这个编辑器窗格本身是否是可编辑的,这个编辑器窗格显示的 HTML 文件是不一样的。图 22-12 中的上图显示了在编辑器窗格不可编辑时 HTML 文件的样子,图 22-12 中的下图显示了在编辑器窗格可编辑时这个文件的样子。

例 22-8 列出了图 22-12 所示的应用程序的代码。

例 22-8 用一个编辑器窗格来显示 HTML 文件

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JFrame {
    private JEditorPane editorPane = new JEditorPane();

    public Test() {
        Container contentPane = getContentPane();

        String url = "file:" + System.getProperty("user.dir") +
            System.getProperty("file.separator") +
            "java.util.Hashtable.html";

        editorPane.setEditable(false);

        try {
            editorPane.setPage(url);
        }
        catch (Exception ex) { ex.printStackTrace(); }

        contentPane.add(new ControlPanel(), BorderLayout.NORTH);
        contentPane.add(new JScrollPane(editorPane),
            BorderLayout.CENTER);
    }

    class ControlPanel extends JPanel {
        private JCheckBox edit = new JCheckBox("Editable");

        public ControlPanel() {
            add(edit);

            edit.addActionListener(new ActionListener() {
                public void actionPerformed ( ActionEvent e ) {
                    editorPane.setEditable(edit.isSelected());
                }
            });
        }

        public static void main(String args[]) {
            GJApp.launch ( new Test(),
                "JEditorPane", 300, 300, 650, 450 );
        }
    }
}
```

这个应用程序创建一个编辑器窗格、一个复选框和一个 HTML 文件的 URL。把这个编辑器窗格的可编辑性设置为 false,而且把一个动作监听器添加到这个复选框中。这个监听器根据复选框选中与否来设置编辑器窗格是否可编辑。

组件总结 22-4 总结了 JEditorPane 组件。

组件总结 22-4 JEditorPane

模型： javax.swing.text.PlainDocument
UI 代表： javax.swing.plaf.basic.BasicEditorPaneUI
绘制器： ——
编辑器： ——
激发的事件：HyperlinkEvents
替代： ——
类图： ——

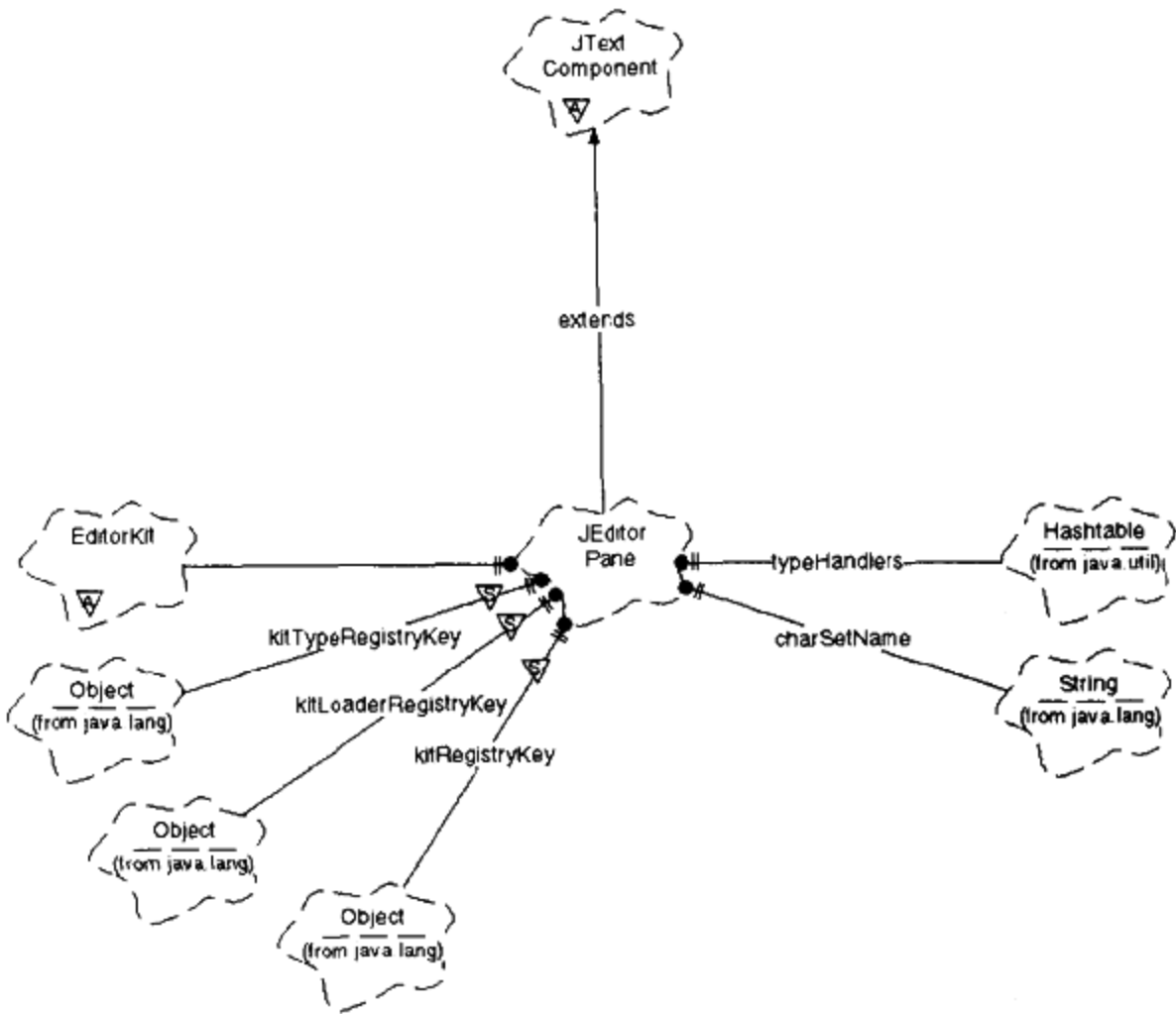


图 22-13 JEditorPane 的类图

JEditorPane 类扩展 JTextComponent 并维护对编辑器工具包的一个私有引用。JEditorPane 使用一个哈希表来使编辑器工具包与内容类型相等。

22.4.1 JEditorPane 属性

表 22-6 列出了由 JEditorPane 维护的属性。

表 22-6 JEditorPane 属性

属性名	数据类型	属性类型 ^①	访问 ^②	缺省值 ^③
editorKit	EditorKit	B	SG	PlainEditorKit
editorKitForContentType	EditorKit	I	SG	—
page	String/URL	B	SG	—
text	String	S	SG	null

① B = 关联的(激发 PropertyChangeEvent)/C = 受约束的/I = 索引的/S = 简单的/Ch = 激发 ChangeEvent
② C = 可在创建时设置/G = 获取方法/S = 设置方法
③ I&F = 与界面样式有关

editorKit——一个编辑器窗格的当前编辑器工具包。编辑器窗格可以把编辑器工具包与不同的内容类型相关联,参见 `editorKitForContentType` 属性。

editorKitForContentType——与一个指定内容类型相关联的编辑器工具包。

page——当前显示的 HTML 文件的 URL。如果一个编辑器窗格当前没有显示一个 URL,则这个属性是 `null`。

text——在一个编辑器窗格中显示的文本。设置文本时,传送给 `setText()` 的这个字符串必须是当前编辑器窗格中内容类型的格式。

Swing 提示

编辑器窗格和编辑器工具包

所有的 Swing 编辑组件都有一个编辑器工具包,但是用于编辑器窗格和文本窗格的编辑器工具包的作用比用于单行文本域和多行文本域的编辑器工具包的作用要大。例如编辑器窗格维护一组编辑器工具包,这些编辑器工具包可插入到不同内容类型的编辑器窗格中。而且,单行文本域和多行文本域的视图由组件的界面样式来创建,而编辑器窗格和文本窗格的视图由组件的编辑器工具包来创建。

22.4.2 JEditorPane 事件

当鼠标在编辑器窗格(显示 HTML 文档的)中的一个超链接上按下时,将激发一个超链接事件。`javax.swing.event` 包定义一个 `HyperlinkListener` 接口和一个 `HyperlinkEvent` 类来处理超链接事件。接口总结 22-1 总结了 `HyperlinkListener` 接口。

接口总结 22-1 HyperlinkListener

```
public abstract void hyperlinkUpdate( HyperlinkEvent )
```

`HyperlinkListener` 接口只定义了一个方法,当激活了 HTML 的超链接时将调用这个方法。`HyperlinkListener.hyperlink` 方法以一个 `HyperlinkEvent` 实例为参数,类总结表 22-4 总结了 `HyperlinkEvent`。

类总结表 22-4 HyperlinkEvent

扩展: `java.util.EventObject`

1. 构造方法

```
public HyperlinkEvent( Object source, HyperlinkEvent.EventType, URL )
```

```
public HyperlinkEvent( Object source, HyperlinkEvent.EventType, URL, String )
```

用事件源、事件类型和一个 URL 来构造超链接事件,还可以用一个描述事件的字符串来构造超链接事件。

2. 方法

```
public String getDescription()
```

```
public HyperlinkEvent.EventType getEventType()
```

```
public URL getURL()
```

`HyperlinkEvent` 提供了用于获得超链接的 URL、事件类型和描述的访问方法(如上所示)。

图 22-14 所示的应用程序说明了对激活超链接所做出的最通常的反应:把与这个超链接相

关联的 URL 装载到一个编辑器窗格中。上图显示了 All Packages 超链接正在被激活,下图显示了与这个超链接相关联的 HTML 页面。

例 22-9 列出了图 22-14 所示的应用程序的代码。

例 22-9 用超链接监听器来装载 URL

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.IOException;

public class Test extends JFrame {
    private JEditorPane editorPane = new JEditorPane();

    public Test() {
        Container contentPane = getContentPane();
        String url = "file:" + System.getProperty("user.dir") +
            System.getProperty("file.separator") +
            "java.util.Hashtable.html";

        try {
            editorPane.setPage(url);
        }
        catch(IOException ex) { ex.printStackTrace(); }

        contentPane.add(new JScrollPane(editorPane),
            BorderLayout.CENTER);

        editorPane.setEditable(false);

        editorPane.addHyperlinkListener ( new HyperlinkListener() {
            public void hyperlinkUpdate ( HyperlinkEvent e ) {
                try {
                    editorPane.setPage(e.getURL());
                }
                catch(IOException ex) { ex.printStackTrace(); }
            }
        });

        public static void main(String args[]) {
            GJApp.launch ( new Test(),
                "JEditorPane", 300, 300, 450,300 );
        }
    }
}
```

22.4.3 JEditorPane 类总结

类总结 22-5 类总结了 JEditorPane 类。

类总结 22-5. JEditorPane a

扩展: JEditorComponent

1. 构造方法

public JEditorPane()

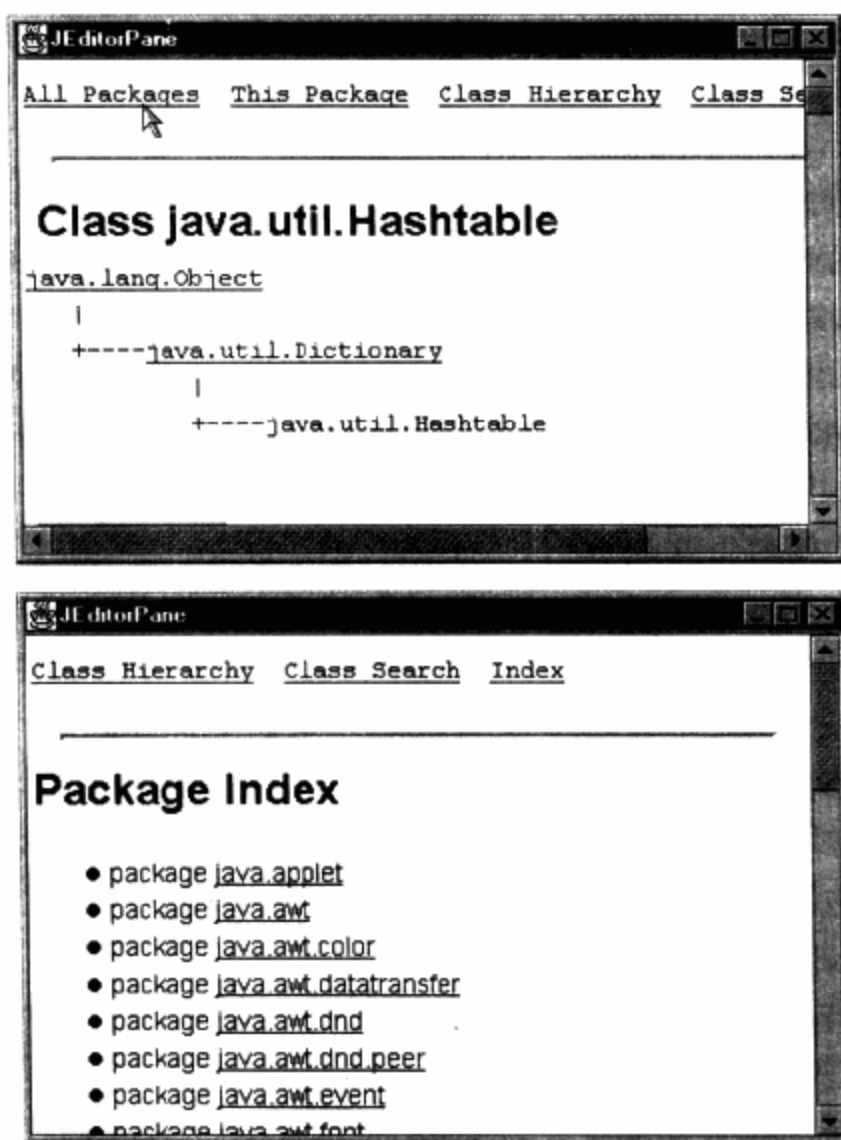


图 22-14 用超链接监听器来装载 URL

```
public JEditorPane( String url ) throws IOException
public JEditorPane( String contentType, String Text )
public JEditorPane( URL ) throws IOException
```

可以用一个 URL 或代表一个 URL 的字符串来创建编辑器窗格,还可以用代表内容类型的字符串和要在编辑器窗格中显示的初始文本来创建编辑器窗格。

2. 方法

(1) 文本/替换选取

```
public String getText()
public void setText( String )
public void replaceSelection( String )
```

上面列出的方法提供了访问编辑器窗格的文本和替换当前选取的文本的能力。传送给 setText 方法的字符串的格式必须与编辑器窗格当前的内容类型相匹配。

replaceSelection 方法用指定的字符串替换当前选取的文本。如果在调用这个方法时没有选取文本,则在当前脱字符的前面插入这个指定的文本。如果这个指定的文本是 null,则删去选取的文本。

(2) 编辑器工具包/内容类型

```
public static void registerEditorKitForContentType( String type, String classname )
public static void registerEditorKitForContentType( String type, String classname, ClassLoader )
public static EditorKit createEditorKitForContentType( String type )
protected EditorKit createDefaultEditorKit()
public final EditorKit getEditorKit()
public EditorKit getEditorKitForContentType( String )
```

```

public void setEditorKit( EditorKit )
public void setEditorKitForContentType( String, EditorKit )

public final String getContentType()
public final void setContentType( String )

```

上面所列的第一组方法向用一个与编辑器工具包对应的类名登记一种内容类型。JEditorPane 维护一张内容类型的哈希表,这个内容类型被作为字符串和编辑器工具包来存储。createEditorKitForContentType 方法搜索哈希表,以便找到与指定内容类型对应的编辑器工具包。

createDefaultEditorKit 方法创建 PlainEditorKit 的一个实例。通过创建 WrappedPlainView 类型的视图,使扩展 DefaultEditorKit.JEditorPane.PlainEditorKit 的一个 DefaultEditorKit 内部类就像编辑器窗格的一个视图库。getEditorKit 方法在编辑器工具包是 null 时创建一个缺省的编辑器工具包并返回这个编辑器窗格的编辑器工具包。

如果以前已经为内容类型指定了一个编辑器工具包,则这个编辑器工具包从 getEditorKitForContentType 方法返回。如果没有编辑器工具包与指定的内容类型相关联,则返回这个内容窗格的编辑器工具包。

getContentType 方法返回配置编辑器窗格的内容类型。setContentType 把一个编辑器的内容类型设置为指定的字符串,引起这个编辑器窗格用相应的编辑器工具包来配置自己。

(3) 页/超链接事件

```

public URL getPage()

public void setPage( String ) throws IOException
public void setPage( URL ) throws IOException

public synchronized void addHyperlinkListener( HyperlinkListener )
public synchronized void removeHyperlinkListener( HyperlinkListener )
public void fireHyperlinkUpdate( HyperlinkEvent )

```

setPage 方法装入指定 URL 的内容,并且把这个编辑器窗格的内容类型设置为 HTML(如果这个编辑器窗格还没有设置内容类型的话)。getPage 方法返回当前在一个编辑器窗格中显示的 URL。如果指定的 URL 无效,则 setPage 将弹出一个 IOException 异常信息,如果 URL 不是当前在编辑器窗格中显示的 URL,则 getPage 将返回一个 null。

上面所列的最后一组方法添加或删除超链接监听器并把超链接事件发送给已登记的监听器。要了解处理超链接事件的样例,请参见 22.4.2 节“JEditorPane 事件”。

(4) Scrollable 接口

```

public boolean getScrollableTracksViewportHeight()
public boolean getScrollableTracksViewportWidth()

```

如果编辑器窗格的视口高度在编辑器窗格的最大值和最小值之间,则 getScrollableTracksViewportHeight 方法返回 true,指示编辑器窗格的高度应保持和其视口高度一致。

如果编辑器窗格的视口宽度在其最大值和最小值之间,则 getScrollableTracksViewportWidth 方法返回 true,指示编辑器窗格的宽度应保持和其视口宽度一致。

(5) 访问性/插入式界面样式

```

public AccessibleContext getAccessibleContext()
public String getUIClassID()

```

与所有其他的 Swing 组件一样,JEditorPane 提供了可以访问它的可访问相关内容的方法 getAccessibleContext。与其他的轻量 Swing 组件一样,JEditorPane 实现了 getUIClassID(),getUIClassID() 返回字符串 EditorPane,这个字符串标识单行文本域的 UI 代表的类。

(6) 其他方法

```
protected InputStream getStream( URL ) throws IOException
public Dimension getPreferredSize()
public boolean isManagingFocus()
protected String  paramString()
protected void processComponentKeyEvent( KeyEvent )
public void read( InputStream, Object description ) throws IOException
public void scrollToReference( String )
```

重载 `getPreferredSize` 方法以考虑当编辑器窗格的视口比编辑器窗格的最小尺寸小的情况,在这种情况下,它返回编辑器窗格的最小大小。

与 `JTextField` 和 `JTextArea` 一样, `JEditorPane` 重载 `isManagingFocus` 以便在编辑器窗格中按下 `Tab` 键将不会把焦点移到下一个组件上。

`read` 方法用一个 `HTML` 编辑器工具包和 `HTML` 文档来配备编辑器窗格(如果这个描述是 `HTMLDocument` 的一个实例)。

22.5 JEditorPane

文本窗格 (`JEditorPane` 类的实例)是 `Swing` 文本组件的凯迪拉克。`JEditorPane` 扩展 `JEditorPane`, 因此, 文本窗格继承了显示不同类型的能力。`JEditorPane` 还增加了两个有意义的功能: 嵌入图标和组件并标记具有属性的内容。

22.5.1 嵌入图标和组件

`JEditorPane` 类提供两种插入图标和组件的方法: `insertComponent(Component)` 和 `insertIcon(Icon)`。图 22-5 所示的应用程序使用这些方法来把一个颜色选取器和一个图像图标插入到一个文本窗格中, 如上图所示。图 22-5 的下图显示了在操纵了颜色选取器以强调放在文本窗格的组件是可用的以后, 这个应用程序的样子。这个图像图标是一个弹跳的小球(一个动画的 `GIF` 文件)。有关图像图标和多帧图像的更多信息, 请参见 5.2.1 节“动画图像图标”。

例 22-10 列出了图 22-15 所示的应用程序的代码。

```
import java.io.File;
import javax.swing.*;
import java.awt.*;
```

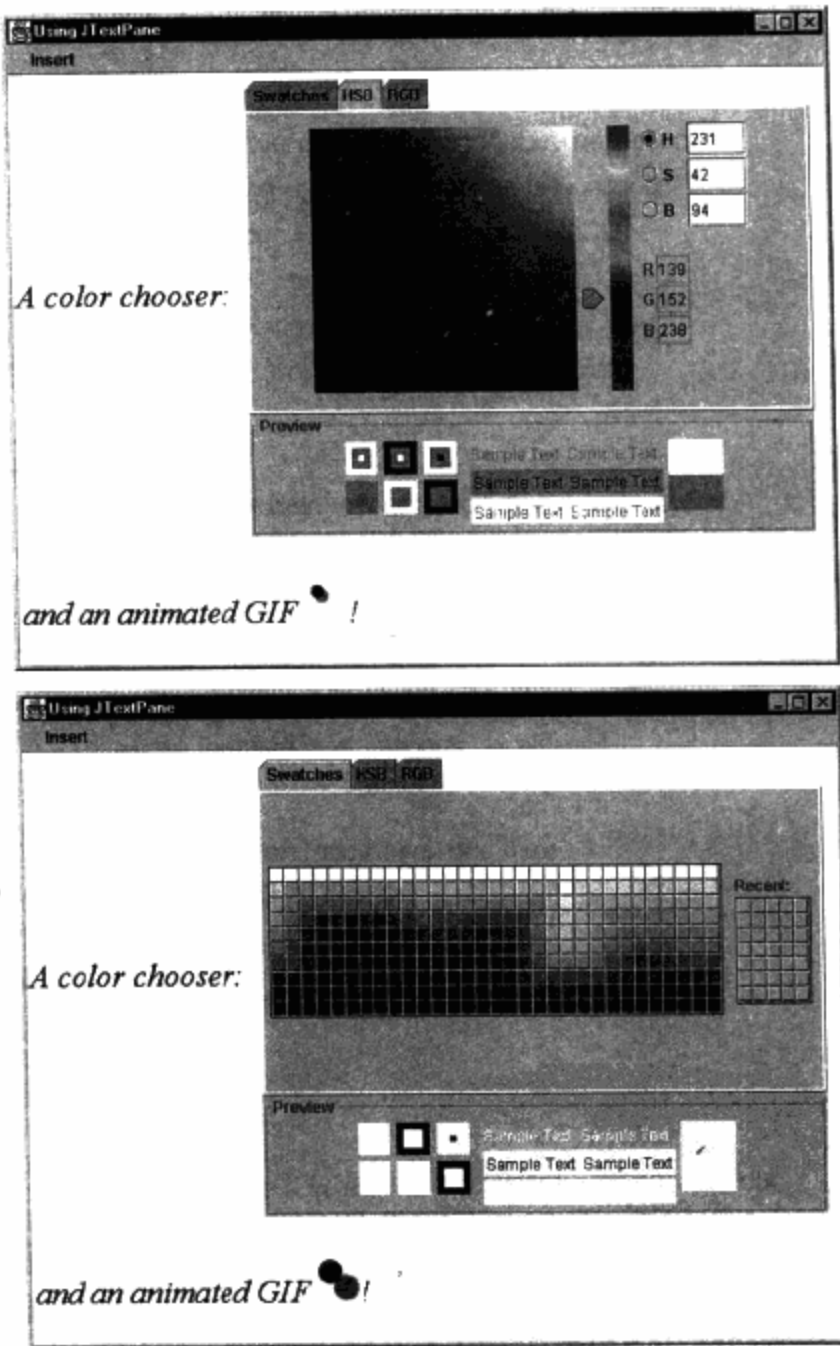


图 22-15 在 `JEditorPane` 中插入组件和图标
例 22-10 文本窗格里的组件和图标

```

import java.awt.event.*;

public class Test extends JFrame {
    private JFileChooser chooser = new JFileChooser();
    private JTextPane textPane = new JTextPane();

    public Test() {
        Container contentPane = getContentPane();
        JMenuBar menuBar = new JMenuBar();
        JMenu insertMenu = new JMenu("Insert");
        JMenuItem imageItem = new JMenuItem("image");
        chooserItem = new JMenuItem("color chooser");

        insertMenu.add(imageItem);
        insertMenu.add(chooserItem);

        menuBar.add(insertMenu);
        setJMenuBar(menuBar);

        textPane.setFont( new Font("Serif", Font.ITALIC, 24));

        contentPane.add(textPane, BorderLayout.CENTER);

        chooserItem.addActionListener( new ActionListener() {
            public void actionPerformed ( ActionEvent e ) {
                JColorChooser chooser = new JColorChooser();
                chooser.setMaximumSize(
                    chooser.getPreferredSize());
                textPane.insertComponent( chooser );
            }
        });

        imageItem. addActionListener( new ActionListener() {
            public void actionPerformed ( ActionEvent e ) {
                int option =
                    chooser.showDialog( Test.this, "Pick An Image");

                if( option == JFileChooser.APPROVE_OPTION ) {
                    File file = chooser.getSelectedFile();

                    if ( file != null ) {
                        textPane.insertIcon( new ImageIcon(
                            file.getPath()));
                    }
                }
            }
        });
    }

    public static void main(String args[] ) {
        GJApp.launch ( new Test(),
            "Using JTextPane", 300, 300, 450, 300 );
    }
}

```

这个应用程序包含一个 Insert 菜单,它有两个菜单项,这两个菜单项用于插入一个图像或一个颜色选取器。这两个菜单项都有一个动作监听器,这个动作监听器用来插入相应的对象。颜色选取器的监听器把它的最大尺寸设置为首选尺寸以限制它的大小。然后,这个监听器调用 `JTextPane.insertComponent` 方法来把这个颜色选取器添加到这个文本窗格中。

22.5.2 用属性标记内容

文本窗格有带风格的编辑器工具包,它提供一组设置常用属性的动作,这些属性如黑体、斜体、下划线、删除线等。StyledEditorKit 包含一个缺省动作数组,这个数组被添加到来自 StyledEditorKit 的超类的动作中:

```
// From StyledEditorKit.java;
public Action[] getActions() {
    return TextAction.argumentList( super.getActions(), this.defaultActions);
}
```

这个文本窗格从它们的编辑器工具包里获取它们的动作,因此,从 JTextPane.getActions() 返回的动作等于 StyledEditorKit 的动作和 StyledEditorKit 的超类的动作总和。

StyledEditorKit 的缺省动作定义如下:

```
//From StyledEditorKit.java
private static final Action[] defaultActions = {
    new FontFamilyAction("font-family-SansSerif", "SansSerif"),
    new FontFamilyAction("font-family-Monospaced", "Monospaced"),
    new FontFamilyAction("font-family-Serif", "Serif"),
    new FontSizeAction("font-size-8", 8),
    new FontSizeAction("font-size-10", 10),
    new FontSizeAction("font-size-12", 12),
    new FontSizeAction("font-size-14", 14),
    new FontSizeAction("font-size-16", 16),
    new FontSizeAction("font-size-18", 18),
    new FontSizeAction("font-size-24", 24),
    new FontSizeAction("font-size-36", 36),
    new FontSizeAction("font-size-48", 48),
    new AlignmentAction("left-justify",
        StyleConstants.ALIGN_LEFT),
    new AlignmentAction("center-justify",
        StyleConstants.ALIGN_CENTER),
    new AlignmentAction("right-justify",
        StyleConstants.ALIGN_RIGHT),
    new BoldAction();
    new ItalicAction();
    new UnderlineAction();
};
```

所有的缺省 StyledEditorKit 动作都作用于当前的选取,例如,选取文本然后激发风格编辑器工具包的黑体动作将为选取的文本设置黑体属性,接着重新绘制这个选取的文本。如果在激发动作时没有文本被选取,则这个动作作用于这个文本窗格的输入属性上,即接着插入的文本将应用这些属性。

所有这些的结果是从 JTextPane.getActions() 返回的动作数组包含上面所列的数组中的动作 (DefaultEditorKit 的大量的动作除外)。图 22-16 所示的应用程序利用了这个特性来用属性标记文本窗格内的内容。

图 22-16 所示的应用程序提供了一个用于剪切、拷贝、粘贴和设置黑体、斜体和下划线的菜单栏和工具条,这个菜单栏还允许设置字体集和字体大小。图 22-16 上面的图片显示这个菜单结构,下面的图片显示在用一些属性标记了某些文本后这个文本窗格的样子。

这个应用程序创建一个 JTextPane 实例和一个字符串数组,这个字符串数组代表剪切、拷贝

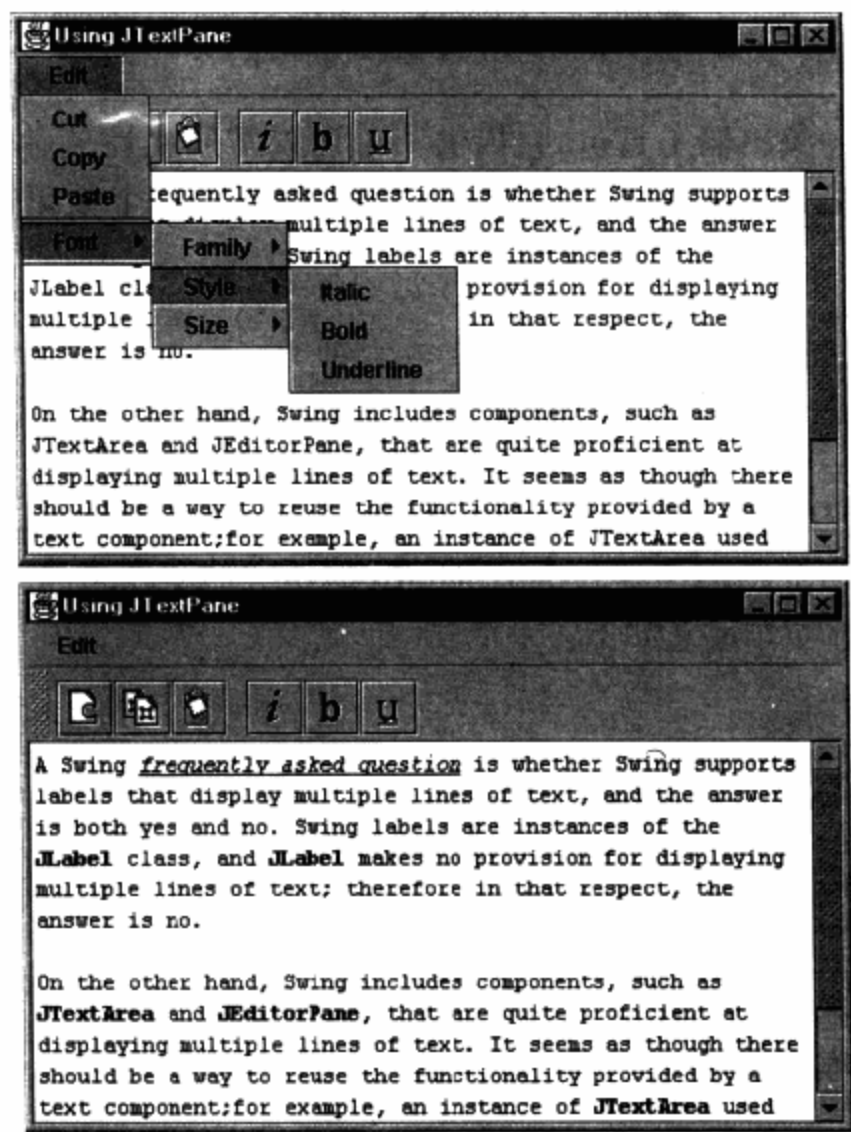


图 22-16 在文本窗格中设置属性

和粘贴动作及在这个菜单中显示的名字和这个工具条的 GIF 文件名。

```
public class Test extends JFrame {
    private JTextPane textPane = new JTextPane();
    ...
    private String[] cutCopyPasteActionNames = new String[] {
        DefaultEditorKit.cutAction, "Cut", "cut.gif",
        DefaultEditorKit.copyAction, "Copy", "copy.gif",
        DefaultEditorKit.pasteAction, "Paste", "paste.gif",
    };
    ...
}
```

为在这个应用程序的菜单栏和工具条中的其他动作创建了三字符串数组。动作名称(如 font-family-SansSerif)是从前面列出的 StyledEditorKit 的缺省动作数组中获得的。值得注意的是 DefaultEditorKit 为其名字和动作提供了一些 public 常量,但 StyledEditorKit 没有提供。

```
...
private String[] familyActionNames = new String[] {
    "font-family-SansSerif", "SanSerif",
    "font-family-Monospaced", "Monospaced",
    "font-family-Serif", "Serif",
};
private String[] styleActionNames = new String[] {
    "font-italic", "Italic", "italic.gif",
    "font-bold", "Bold", "bold.gif",
    "font-underline", "Underline", "underline.gif",
};
```



```
private String[] sizeActionNames = new String[] {
    "font-size-8", "8", "font-size-10", "10",
    "font-size-12", "12", "font-size-14", "14",
    "font-size-16", "16", "font-size-18", "18",
    "font-size-24", "24", "font-size-36", "36",
    "font-size-48", "48",
};
...
```

这个应用程序提供了一个 `populate` 方法,这个方法配置上面讨论的字符串数组中的菜单栏和工具条。这个应用程序把这个文本窗格的动作和它们的名字存储在一个哈希表中,并提供一个 `getAction` 方法,这个方法(给定一个名字)返回一个动作。有关通过名字访问一个文本组件的动作的更多信息,请参见 21.2.2 节“动作和编辑器工具包”。

对每个字符串数组而言,可以用这个应用程序的 `getAction` 方法来获得与这个名字相关联的动作,这个动作被添加到菜单栏、工具条中或同时添加到菜单栏和工具条中。

```
...
private void populate() {
    JMenu editMenu = new JMenu("Edit");
    fontMenu = new JMenu("Font");
    styleMenu = new JMenu("Style");
    sizeMenu = new JMenu("Size");
    familyMenu = new JMenu("family");

    for (int i = 0; i < familyActionNames.length; ++i) {
        Action action = getAction(familyActionNames[i]);
        if (action != null) {
            JMenuItem item = familyMenu.add(action);
            item.setText(familyActionNames[i]);
        }
    }

    for (int i = 0; i < sizeActionNames.length; ++i) {
        Action action = getAction(sizeActionNames[i]);
        if (action != null) {
            JMenuItem item = sizeMenu.add(action);
            item.setText(sizeActionNames[i]);
        }
    }

    for (int i = 0; i < cutCopyPasteActionNames.length; ++i) {
        Action action = getAction(cutCopyPasteActionNames[i]);
        if (action != null) {
            JButton button = toolbar.add(action);
            JMenuItem item = editMenu.add(action);

            item.setText(cutCopyPasteActionNames[i]);

            button.setText(null);
            button.setIcon(new ImageIcon(
                cutCopyPasteActionNames[i]));
        }
    }

    editMenu.addSeparator();
    toolbar.addSeparator();

    for (int i = 0; i < styleActionNames.length; ++i) {
        Action action = getAction(styleActionNames[i]);
    }
}
```

```

if (action != null) {
    JButton button = toolbar.add(action);
    JMenuItem item = styleMenu.add(action);

    item.setText(styleActionNames[ ++ i]);

    button.setText(null);
    button.setIcon(
        new ImageIcon(styleActionNames[ ++ i]));
}
}
...
}
...
}

```

例 22-11 列出了图 22-16 所示应用程序的完整代码。

例 22-11 在文本窗格中设置字符属性

```

import java.io.File;
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.FileReader;

public class Test extends JFrame {
    private JTextPane textPane = new JTextPane();
    private JMenuBar menubar = new JMenuBar();
    private JToolBar toolbar = new JToolBar();
    private Hashtable actionTable = new Hashtable();

    private String[] cutCopyPasteActionNames = new String[] {
        DefaultEditorKit.cutAction, "Cut", "cut.gif",
        DefaultEditorKit.copyAction, "Copy", "copy.gif",
        DefaultEditorKit.pasteAction, "Paste", "paste.gif",
    };

    private String[] familyActionNames = new String[] {
        "font-family-SansSerif", "SanSerif",
        "font-family-Monospaced", "Monospaced",
        "font-family-Serif", "Serif",
    };

    private String[] styleActionNames = new String[] {
        "font-italic", "Italic", "italic.gif",
        "font-bold", "Bold", "bold.gif",
        "font-underline", "Underline", "underline.gif",
    };

    private String[] sizeActionNames = new String[] {
        "font-size-8", "8", "font-size-10", "10",
        "font-size-12", "12", "font-size-14", "14",
        "font-size-16", "16", "font-size-18", "18",
        "font-size-24", "24", "font-size-36", "36",
        "font-size-48", "48",
    };
}

```

```

public Test() {
    Container contentPane = getContentPane();
    JScrollPane scrollPane = new JScrollPane(textPane);

    loadActionTable();
    populate();
    readFile();
    setJMenuBar(menuBar);

    contentPane.add(toolbar, BorderLayout.NORTH);
    contentPane.add(new JScrollPane(textPane),
        BorderLayout.CENTER);
}

private void readFile() {
    try {
        textPane.getEditorKit().read(
            new FileReader("text").textPane.getDocument(), 0);
    }
    catch (Exception ex) { ex.printStackTrace(); }
}

private void populate() {
    JMenu editMenu = new JMenu("Edit");
    fontMenu = new JMenu("Font");
    styleMenu = new JMenu("Style");
    sizeMenu = new JMenu("Size");
    familyMenu = new JMenu("Family");

    for (int i = 0; i < familyActionNames.length; ++i) {
        Action action = getAction(familyActionNames[i]);
        if (action != null) {
            JMenuItem item = familyMenu.add(action);
            item.setText(familyActionNames[i]);
        }
    }

    for (int i = 0; i < sizeActionNames.length; ++i) {
        Action action = getAction(sizeActionNames[i]);
        if (action != null) {
            JMenuItem item = sizeMenu.add(action);
            item.setText(sizeActionNames[i]);
        }
    }

    for (int i = 0; i < cutCopyPasteActionNames.length; ++i) {
        Action action = getAction(cutCopyPasteActionNames[i]);
        if (action != null) {
            JButton button = toolbar.add(action);
            JMenuItem item = editMenu.add(action);

            item.setText(cutCopyPasteActionNames[i]);
            button.setText(null);
            button.setIcon(new ImageIcon(
                cutCopyPasteActionNames[i]));
        }
    }

    editMenu.addSeparator();
    toolbar.addSeparator();
}

```

```
for (int i = 0; i < styleActionNames.length; ++i) {
    Action action = getAction(styleActionNames[i]);
    if (action != null) {
        JButton button = toolbar.add(action);
        JMenuItem item = styleMenu.add(action);

        item.setText(styleActionNames[i]);

        button.setText(null);
        button.setIcon(
            new ImageIcon(styleActionNames[i]));
    }
}

fontMenu.add(familyMenu);
fontMenu.add(styleMenu);
fontMenu.add(sizeMenu);
editMenu.add(fontMenu);
menubar.add(editMenu);
}

private void loadActionTable() {
    Action[] actions = textPane.getActions();

    for (int i = 0; i < actions.length; ++i) {
        actionTable.put(actions[i].getValue(Action.NAME),
            actions[i]);
    }
}

private Action getAction(String name) {
    return (Action)actionTable.get(name);
}

public static void main(String args[]) {
    GJApp.launch(new Test(),
        "Using JTextPane", 300, 300, 450, 300);
}
}
```

组件总结 22-5 总结了 JTextPane 组件

组件总结 22-5 **JTextPane**

模型:	javax.swing.text.StyledDocument
UI 代表:	javax.swing.plaf.basic.Basic TextPaneUI
绘制器:	——
编辑器:	——
激发的事件:	——
替代:	——
类图:	见图 22-17

JTextPane 是 JEditorPane 的一个扩展,它除了一个 UI 代表类 ID 外,没有其他的成员变量。JTextPane 缺少成员变量归因于 JTextPane 依赖其编辑器工具包和文档来提供功能。例如, JTextPane 把设置属性交给它的文档来处理,这个文档的缺省值是 DefaultStyledDocument 的实例。

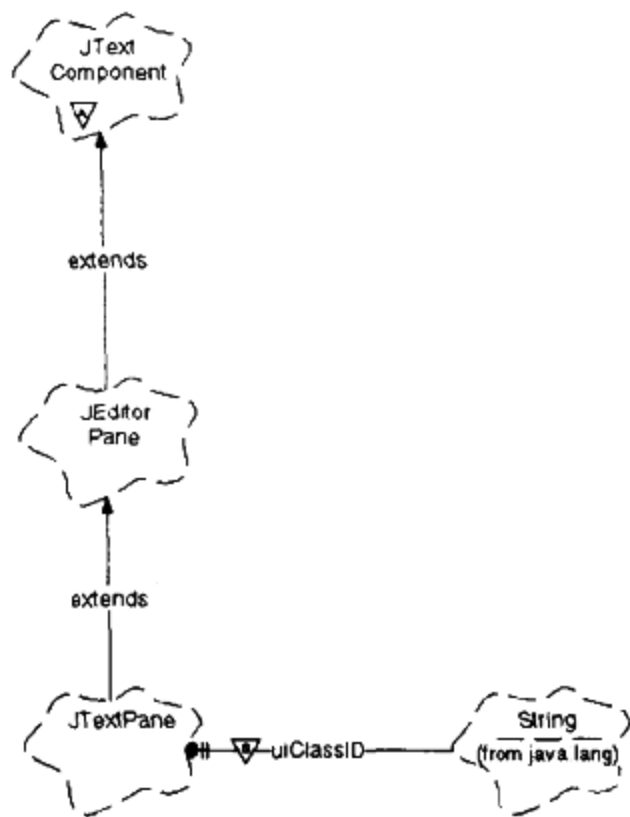


图 22-17 JTextPane 类图

22.5.3 JTextPane 属性

表 22-7 列出了由 JTextPane 类维护的属性。

表 22-7 JTextPane 属性

属性名	数据类型	属性类型 ⁽¹⁾	访问 ⁽²⁾	缺省值 ⁽³⁾
characterAttributes	AttributeSet	S	SG	—
inputAttributes	AttributeSet	S	SG	—
logicalStyle	Style	I	SG	—
paragraphAttributes	AttributeSet	S	SG	—
style	Style	I	G	—

① B = 关联的(激发 PropertyChangeEvent)/C = 约束的/I = 索引的/S = 简单的/Ch = 激发 ChangeEvent
② C = 可在创建时设置/G = 获取方法/S = 设置方法
③ I&F = 与界面样式有关

characterAttributes——当前选取的属性集。如果在设置此属性时没有文本被选中,则这个属性被用在接下来要插入的文本中。

inputAttributes——一个属性集,用在下一次要插入的文本中。

logicalStyle——用于段落的一个风格(在字符或段落属性没有被显式地设置时)。

paragraphAttributes——用于设置加字符所在段的段落属性。

style——以前添加到文本窗格中的一个有名字的风格。

22.5.4 JTextPane 类总结

类总结 22-6 总结了 JTextPane 类。

类总结 22-6 JTextPane

扩展: JEditorPane

1. 构造方法

public JTextPane()

```
public JTextPane( StyledDocument )
```

可以用无参数的构造方法来创建文本窗格,这个构造方法把文本窗格的编辑器工具包设置为 `StyledEditorKit` 的一个实例。缺省时,风格编辑器工具包创建文档(它们是 `DefaultStyledDocument` 的实例)和不同类型的视图。

2. 方法

(1) 文档/编辑器工具包/替换选取

```
public void setDocument( Document )
public final void setEditorKit( EditorKit )
public void replaceSelection( String )
```

文本窗格必须有一个 `StyledDocument` 实例作为它的文档,因此 `JTextPane` 重载 `setDocument()`,当指定的文档不是一个 `StyledDocument` 实例时,它将弹出一个非法参数的异常信息。文本窗格还必须有一个 `StyledEditorKit` 实例作为它的编辑器工具包,因此它重载了 `setEditorKit()`,如果指定的编辑器工具包不是 `StyledEditorKit` 的一个实例,则它将弹出一个非法参数的异常信息。

`replaceSelection` 方法在文本窗格是可编辑的时候用指定的文本来替换当前选取的文本。如果调用此方法时没有文本被选中,则在加字符位置插入这个文本。

(2) 风格文档/风格编辑器工具包

```
protected EditorKit createDefaultEditorKit()
protected final StyledEditorKit getStyledEditorKit()
public StyledDocument getStyledDocument()
public void setStyledDocument( StyledDocument )
```

上面所列的前两个保护方法创建 `JTextPane` 的缺省编辑器工具包(一个 `StyledEditorKit` 实例),并提供访问风格编辑器工具包的方法。

`getStyledDocument` 和 `setStyledDocument` 方法分别调用 `super.getDocument()` 和 `super.setDocument()`。`getStyledDocument` 方法允许把文本窗格的文档作为 `StyledDocument` 来访问,而不用先从 `getDocument()` 返回再处理这个文档。

(3) 风格

```
public Style addStyle( String styleName, Style parent )
public Style getStyle( String styleName )
public void removeStyle( String styleName )
public Style getLogicalStyle()
public void setLogicalStyle( Style )
```

文本窗格的文档维护一个风格集。上面所列的前三个方法提供了访问这些的风格方法。`addStyle` 创建并返回一个已命名的风格,把这个风格添加到指定的父风格中。

`setLogicalStyle` 方法在字符或段落的属性还没有被显式地设置时使用。当调用 `setLogicalStyle` 方法时,这个方法为加字符处的段落设置逻辑风格。`getLogicalStyle` 返回当前加字符处段落的逻辑风格。

(4) 图标和组件

```
public void insertComponent( component )
public void insertIcon( Icon )
```

上面所列的方法分别插入一个组件或一个图标以替代当前的选取。如果调用这个方法时没有选取,则在插入发生在当前的加字符处。

(5) 属性

```
public AttributeSet getCharacterAttributes()
```

```
public void setCharacterAttributes( AttributeSet, boolean replace )
public AttributeSet getParagraphAttributes( )
public void setParagraphAttributes( AttributeSet, boolean replace )
public MutableAttributeSet getInputAttributes( )
```

setCharacterAttributes 方法设置当前选中内容的字符属性。如果调用时没有选中内容,则这些属性将应用到随后插入到文本窗格中的内容中。getCharacterAttributes 方法返回当前加字符处的字符的属性。

setParagraphAttributes 方法设置当前选中内容的段落属性。如果调用时没有选中内容,则这些属性将应用到在加字符处的段落中。getParagraphAttributes 方法返回当前加字符所在处的段落属性。

getInputAttributes 方法属性集,这个属性集在当前加字符处插入的内容中使用。

(6) 其他方法

```
public boolean getScrollableTracksViewportWidth( )
public String getUIClassID( )
protected String paramString( )
```

getScrollableTracksViewportWidth 方法返回 true,即文本窗格的宽度总是与它所在的视口的宽度相同。paramString 方法返回一个代表文本窗格的字符串,getUIClassID 返回一个字符串——TextPane,这个字符串标识 JTextPane 实例的 UI 代表。

22.6 AWT 兼容

AWT 没有和 JTextPane 类似的组件。

22.7 本章回顾

在第21章“文本基础”中介绍了所有 Swing 文本组件的基础功能,包括动作、键映射、加字符、加重器等。本章详细介绍了每一个 Swing 文本组件。

本章提供的内容足够一般的使用,但如果要定制组件则远远不够。第23章将介绍 Swing 文本包更高级的功能,包括视图、风格和与风格相关的内容。

第 23 章 定制文本组件

Swing 文本建立在由 `javax.swing.text` 包的类和接口提供的一个复杂的下层构件之上。一般使用 Swing 文本组件（在前两章中介绍）不要求对 Swing 文本包有很深的了解。但如果要定制文本组件，则要对 `javax.swing.text` 包有一个基本的掌握。本章提供了定制通用任务的例子，如彩色文本、设置字符和段落属性、实现定制视图等。

23.1 概览

与其他 Swing 组件一样，文本组件由一个模型（Document 接口的一个实现）和一个 UI 代表（`javax.swing.plaf.basic.BasicTextUI` 类的一个扩展）组成。文本组件还使用一个编辑器工具包（`EditorKit` 类的一个扩展）和一个视图（`View` 类的一个扩展）。图 23-1 示出了一个类图，这个类图说明了 Swing 文本域、这个域模型、UI 代表、编辑器工具包和视图之间的静态关系^①。其他的 Swing 文本组件有类似的类图。

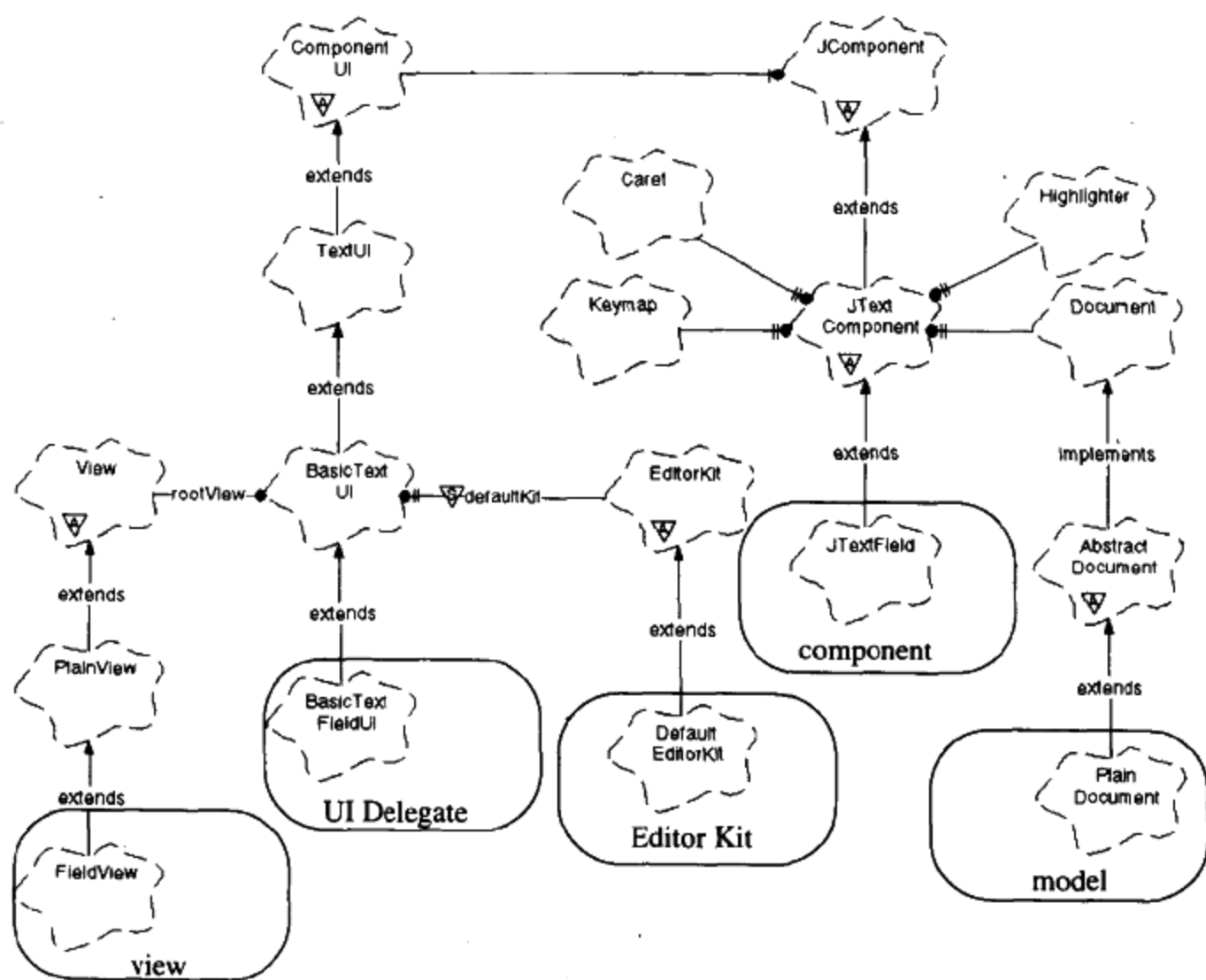


图 23-1 JTextField 和支持类

所有的 Swing 文本组件都扩展 `javax.swing.text.package` 包中的 `JtextComponent` 类。`JTextCompo-`

④ 23.4 节“视图”讨论了视图，所有的文本组件都有一个或多个视图。

nent 的实例维护一张键映射表、加字符、加重器和文档，它们在第 21 章中已介绍过了。

与其他的 Swing 轻量组件一样，JTextComponent 类扩展 JComponent，JComponent 又维护对这个组件的 UI 代表（Javax.swing.plaf.ComponentUI 类的一个扩展）的一个引用。所有的文本组件都有一个 UI 代表，这个 UI 代表扩展 javax.swing.plaf.basic.BasicTextUI 类。BasicTextUI 维护所有文本组件的编辑器工具包和视图。

JTextField 类使用一个编辑器工具包（DefaultEditorKit 的一个实例）和一个视图（FieldView 类的一个实例）。表 23-1 列出了被每个 Swing 文本组件使用的编辑器工具包、文档和视图。

表 23-1 缺省编辑器工具包、文档和视图

名字	编辑器工具包	文档	视图
JTextArea	DefaultEditorKit	PlainDocument	Plain View/ WrappedPlainView
JTextField	DefaultEditorKit	PlainDocument	FieldView
JPasswordField	DefaultEditorKit	PlainDocument	PasswordView
JEditorPane	JEditorPane.PlainEditorKit	PlainDocument	Wrapped PlainView
JTextPane	StyledEditorKit	DefaultStyledDocument	LabelView

所有的文本组件都有一个编辑器工具包，所有的组件使用它们的编辑器工具包来存储动作、创建缺省的文档和读写文档。文本编辑器（JTextPane 和 JEditorPane）还使用它们的编辑器工具包来创建视图。

如果文档没有被显式地指定，则在安装组件的 UI 代表时创建这些文档。BasicTextUI 重载 installUI() 并交给这个组件的编辑器工具包来创建缺省的文档。注意，是编辑器工具包最后创建了缺省文档。

```
// From javax.swing.plaf.basic.BasicTextUI:
public void installUI (JComponent c) {
    if (c instanceof JTextComponent) {
        editor = (JTextComponent) c;
        ...
        Document doc = editor.getDocument ();
        If (doc == null) {
            editor.setDocument (
                getEditorKit (editor) .createDefaultDocument () );
        } else {
            ...
        }
    }
    ...
}
```

文本组件的视图由视图库来创建，视图库由 ViewFactory 接口定义。ViewFactory 只定义了一个方法 Viewcreate (Element)。给定一个元素，^① 视图库就创建一个视图。BasicTextUI 实现 ViewFactory 接口并返回一个 null 视图：

```
// From javax.swing.plaf.basic.BasicTextFieldUI:
public View create (Element element) {
    return null;
}
```

① 有关元素的更多信息，请参见 23.6 节“元素”。

BasicTextUI 的一些扩展重载 create 方法，例如 BasicTextFieldUI 和 BasicTextArea 实现了 create 方法，如下面的程序所示：

```
// From javax.swing.plaf.basic.BasicTextFieldUI:
public View create (Element element) {
    return new FieldView ;
}
// From javax.swing.plaf.basic.BasicTextAreaUI:
public View create (Element elem) {
    JTextComponent c = getComponent () ;
    if (c instanceof JTextArea) {
        JTextArea area = (JTextArea) c ;
        View v ;
        if (area.getLineWrap ()) {
            v =
                new WrappedPlainView (elem, area.getWrapStyleWord ()) ;
        } else {
            v = new PlainView (elem) ;
        }
        return v ;
    }
    return null ;
}
```

BasicTextFieldUI.create () 总是为文本域返回缺省视图的一个 FieldView 实例。它根据文本域是否能换行来返回一个普通视图还是一个可换行的视图。

BasicTextPaneUI 和 BasicEditorPaneUI 没有重载 create 方法，它们的视图由组件的编辑器工具包来创建。

23.2 属性集和风格常量

每个文本组件都有一个文档，每个文档维护一个元素分层。有关文档和元素的更多信息，请参见 23.6 节“元素”。元素维护一个文档中的起始位置和结束位置，并维护一个属性集，这个属性集由位于起始位置和结束位置之间的内容所使用。

属性集由 AttributeSet 接口定义。图 23-2 示出了 AttributeSet 接口的类图。

AttributeSet 接口由 MutableAttributeSet 接口来扩展，MutableAttributeSet 接口定义修改属性集属性的方法。MutableAttributeSet 由 Style 接口来扩展（风格被称作可变的属性集，这些风格在修改它们的属性时将通知变化监听器）。SimpleAttribute

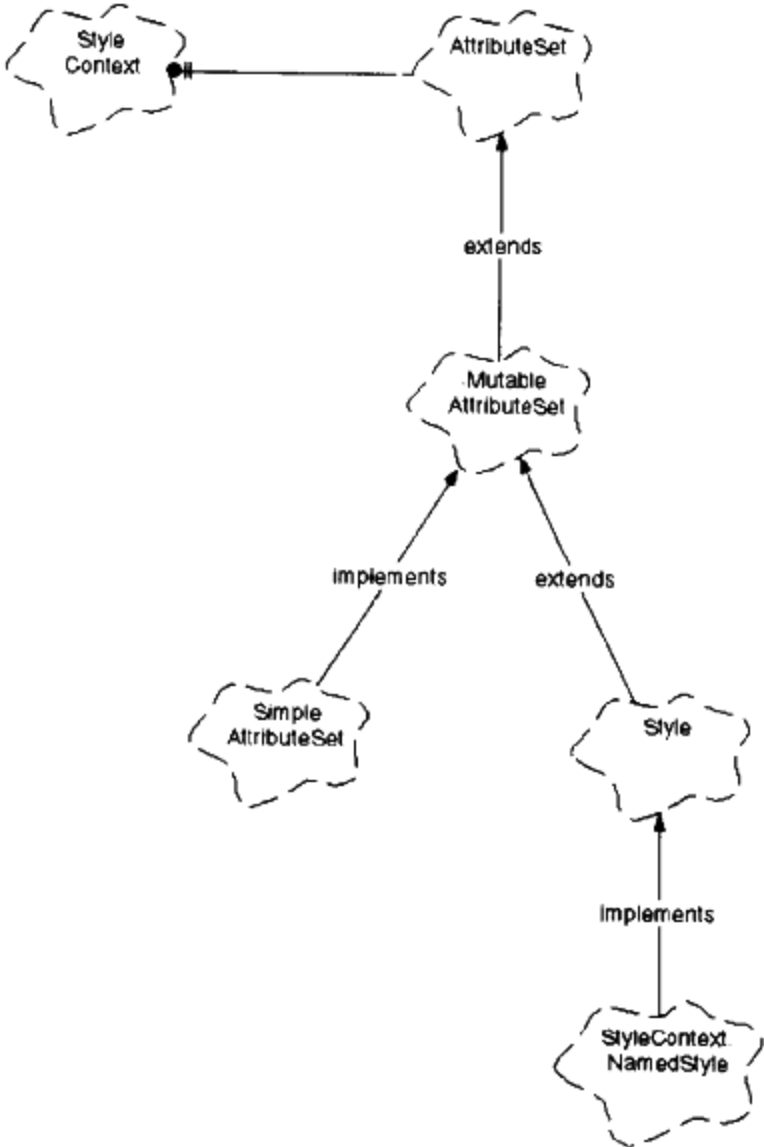


图 23-2 AttributeSet 的类图

类也维护 MutableAttributeSet 接口，StyleContext.NamedStyle 类维护 Style 接口。

图 23-3 所示的应用程序说明了属性集的使用。这个应用程序包含一个文本窗格，并为包含在这个文本窗格模型中的 component 和 container 字符串设置了属性。

例 23-1 列出了图 23-3 所示应用程序的完整代码。

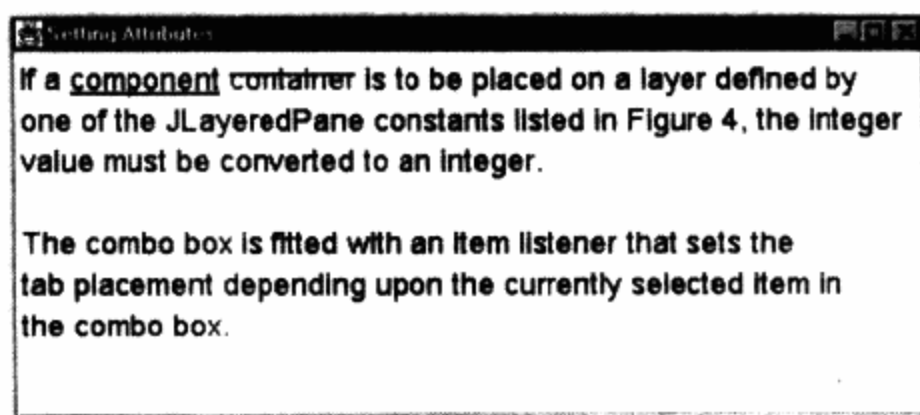


图 23-3 设置属性

例 23-1 使用属性集

```
import java.io.File;
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.FileReader;

public class Test extends JFrame {
    private JTextPane textPane = new JTextPane ();
    private StyledDocument document =
        (StyledDocument) textPane.getDocument ();

    public Test () {
        Container contentPane = getContentPane ();
        readFile ("text.txt");
        setAttributes ();
        textPane.setFont (new Font ("Dialog", Font.PLAIN, 18));
        contentPane.add (new JScrollPane (textPane),
            BorderLayout.CENTER);
    }

    private void setAttributes () {
        SimpleAttributeSet attributes = new SimpleAttributeSet ();
        StyleConstants.setForeground (attributes, Color.blue);
        StyleConstants.setUnderline (attributes, true);
        document.setCharacterAttributes (5, 9, attributes, false);
        StyleConstants.setForeground (attributes, Color.red);
        StyleConstants.setStrikeThrough (attributes, true);
        document.setCharacterAttributes (15, 9, attributes, false);
    }

    private void readFile (String filename) {
        EditorKit kit = textPane.getEditorKit ();
        try {
            kit.read (new FileReader (filename), document, 0);
        }
        catch (Exception ex) {
            ex.printStackTrace ();
        }
    }
}
```

```

|
|
|
public static void main (String args []) {
    GJApp.launch (new Test (),
        "Setting Attributes", 300, 300, 450, 300);
|
|

```

文本窗格的文档是 `StyledDocument` 的一个实例，这个实例提供一个 `setCharacterAttributes` 方法，这个方法把一个属性集应用到这个文档的一个指定范围上。`component` 的前景色被设置为蓝色而且下划线属性被设置为 `true`。`container` 的前景色被设置为红色而且删除线属性被设置为 `true`。

这些属性是在 `StyleConstants` 类的帮助下设置的，`StyleConstants` 类提供了 40 多个 `static` 方法来为一个指定的属性集设置每一个属性，这些属性从黑体和斜体属性到下划线和下标（上标）属性。

与键映射一样，属性集分层地进行解析，也就是说，如果在一个属性集中没有找到某个属性，则搜索这个属性集的原型以找到这个属性。`StyledDocument.setCharacterAttributes()` 的第二个参数决定了是否应该用一个指定的属性来替换文档的属性集，或是否应该把属性添加到当前的属性集中。例 23-1 的例子把第二个参数设置为 `false`。

Swing 提示

风格常量

用属性名字和一个值来指定属性集中的属性。`AttributeSet` 接口定义一个 `addAttribute` 方法：`public void addAttribute (Object name, Object value)`，用它来把一个属性添加到属性集中。`StyleConstants` 类提供人们熟悉的属性的静态设置方法和获取方法，这些人们熟悉的属性如黑体、斜体、下划线、删除线等等。使用设置属性的 `StyleConstants` 确保这些人们熟悉的属性使用的名字是一致的。

23.3 定制动作

21.2 节“动作”介绍了文本组件的动作并说明了如何使用编辑器工具包的缺省动作。经常为定制的文本组件实现定制的动作，像图 23-4 中所示的程序那样。

图 23-4 所示的应用程序包含一个文本窗格并提供了一个菜单，这个菜单用于修改文本窗格选取的那些字符的背景色或加字符所在的那一段落的背景色。图 23-4 中上面的图片说明了设置段落属性的情形，图 23-4 中下面的图片说明了设置字符属性的情形。

这个应用程序实现一个作为内部类的定制动作 (`ForegroundFromChooserAction`)，这个动作使用从颜色调色板中选取的颜色来设置前景色。关于颜色调色板的更多信息，请参见 16.2.2 节。`ForegroundFromChooserAction` 类扩展 `StyleEditorKit.StyledAction`，如下所示。

这个动作维护一个模式 (`CharacterMode` 或 `ParagraphMode`)，当创建一个动作时，将指定一个模式。这个模式确定是把所选取的颜色应用到选取的那些字符中还是把所选取的颜色应用到

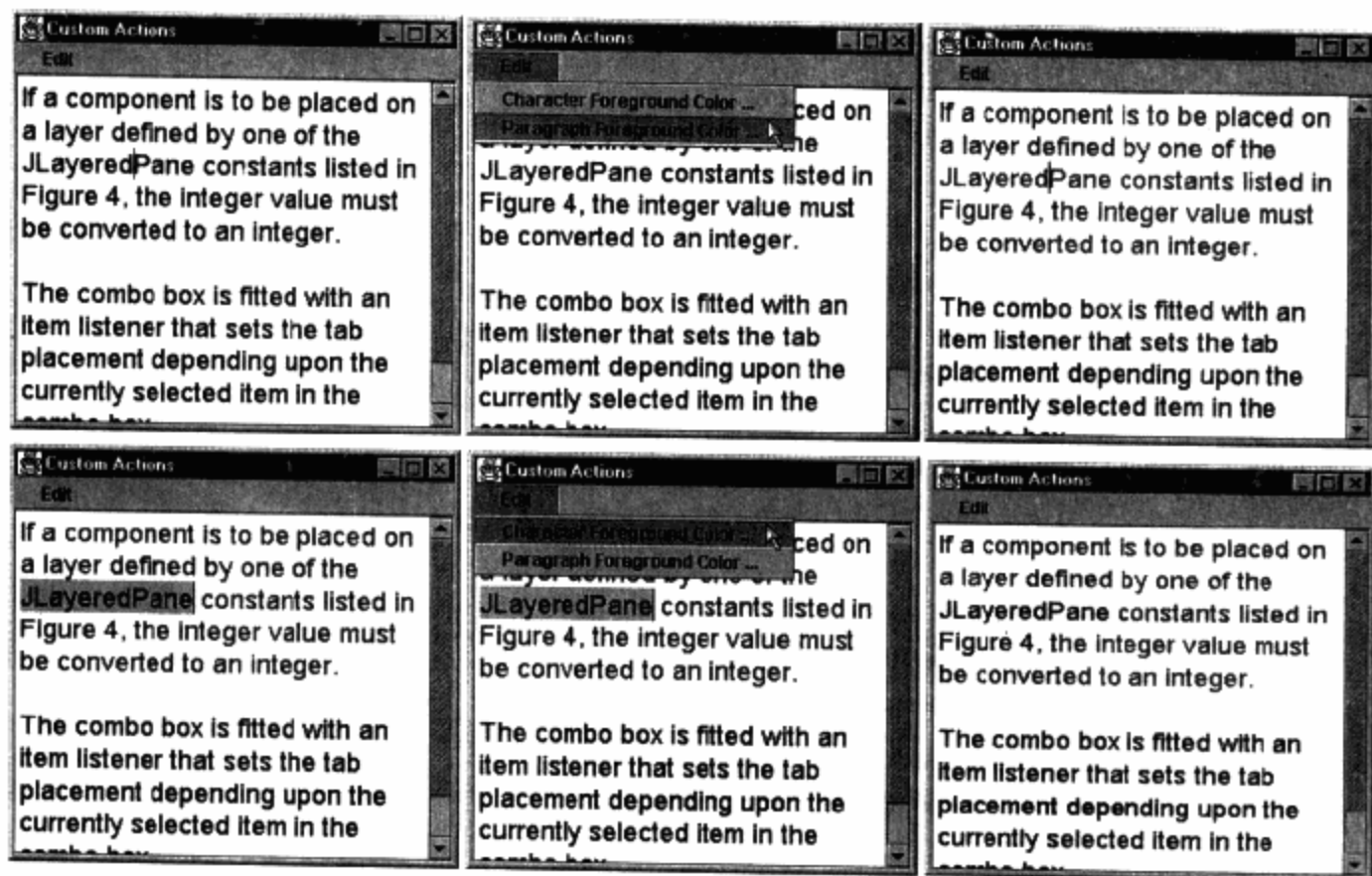


图 23-4 用于设置字符和段落属性的定制动作

加字符所处的段落中。

```
...
public class Test extends JFrame {
    ...
    private int CharacterMode = 0, ParagraphMode = 1;
    ...
    class ForegroundFromChooserAction
        extends StyledEditorKit.StyleTextAction {
        protected Color fg;
        protected JColorChooser chooser = new JColorChooser ();
        protected int mode;

        public ForegroundFromChooserAction (String nm, int mode) {
            super (nm);
            this.mode = mode;
        }
    }
    ...
}
```

actionPerformed 方法通过调用从 StyledEditorKit.StyleTextAction 继承的 getEditor 方法来获得对文本窗格的一个引用。用 JTextPane.setCharacterAttributes () 可获得与文本窗格相关联的字符属性，getCharacterAttributes () 方法返回与当前的选取相关联的属性^①。通过调用 StyleConstants.setForeground 方法来获得这个文本窗格字符属性的前景色，然后用这个前景色来初始化颜色调色板。

如果从这个颜色调色板里选取了一种颜色，则创建一个 SimpleAttributeSet 实例，并用方法 StyleConstants.setForeground 来设置前景属性。把这个简单的属性传递给 setCharacterAttributes 或 setParagraphAttributes 以便设置前景色。如果没有选取颜色，则用缺省的编辑器工具包来发出一声蜂鸣声。

① 如果没有选取，则返回输入属性。

```

...
public void actionPerformed (ActionEvent e) {
    JEditorPane editor = getEditor (e);
    if (editor != null) {
        AttributeSet attributes =
            textPane.setCharacterAttributes ();
        Color c =
            StyleConstants.setForeground (attributes);
        Color fg = chooser.showDialog (Test.this,
            "Choose Color for Text",
            c == null ? Color.black : c);
        if (fg != null) {
            MutableAttributeSet attr =
                new SimpleAttributeSet ();
            StyleConstants.setForeground (attr, fg);
            if (mode == CharacterMode)
                setCharacterAttributes (editor, attr, false);
            else
                setParagraphAttributes (editor, attr, false);
        }
        else {
            Toolkit.getDefaultToolkit ().beep ();
        }
    }
}
...
}

```

例 23-2 列出了图 23-4 所示应用程序的完整代码。

例 23-2 实现定制动作

```

import java.io.File;
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.FileReader;

public class Test extends JFrame {
    private JTextPane textPane = new JTextPane ();
    private StyledDocument document =
        (StyledDocument) textPane.getDocument ();
    private StyledEditorKit kit =
        (StyledEditorKit) textPane.getEditorKit ();
    private JColorChooser chooser = new JColorChooser ();
    private int CharacterMode = 0, ParagraphMode = 1;

    public Test () {
        Container contentPane = getContentPane ();
        readFile ("text.txt");
        textPane.setFont (new Font ("Dialog", Font.PLAIN, 18));
    }
}

```



```

        contentPane.add (new JScrollPane (textPane),
                        BorderLayout.CENTER);

        setJMenuBar (createMenuBar ());
    }

    private JMenuBar createMenuBar () {
        JMenuBar menuBar = new JMenuBar ();
        JMenu editMenu = new JMenu ("Edit");

        editMenu.add (new ForegroundFromChooserAction (
            "Character Foreground Color ... ",
            CharacterMode));

        editMenu.add (new ForegroundFromChooserAction (
            "Paragraph Foreground Color ... ",
            ParagraphMode));

        menuBar.add (editMenu);
        return menuBar;
    }

    private void readFile (String filename) {
        try {
            kit.read (new FileReader (filename), document, 0);
        }
        catch (Exception ex) {
            ex.printStackTrace ();
        }
    }

    public static void main (String args []) {
        GJApp.launch (new Test (),
            "Custom Actions", 300, 300, 650, 275);
    }

    class ForegroundFromChooserAction
        extends StyledEditorKit.StyledTextAction {

        protected Color fg;
        protected JColorChooser chooser = new JColorChooser ();
        protected int mode;

        public ForegroundFromChooserAction (String nm, int mode) {
            super (nm);
            this.mode = mode;
        }

        public void actionPerformed (ActionEvent e) {
            JEditorPane editor = getEditor (e);

            if (editor != null) {
                AttributeSet attributes =
                    textPane.setCharacterAttributes ();
                Color c =
                    StyleConstants.setForeground (attributes);

                Color fg = chooser.showDialog (Test.this,
                    "Choose Color for Text",
                    c == null ? Color.black : c);

                if (fg != null) {
                    MutableAttributeSet attr =
                        new SimpleAttributeSet ();
                    StyleConstants.setForeground (attr, fg);
                }
            }
        }
    }

```

```
if (mode == CharacterMode)
    setCharacterAttributes (editor, attr, false);
else
    setParagraphAttributes (editor, attr, false);
textPane .setCaretPosition (
    textPane.getSelectionStart ());
| else |
    Toolkit.getDefaultToolkit ().beep ();
```

23.4 视图

Swing 组件实现一个模式-视图-控制器的体系结构，这个体系结构把一个组件的数据（模型）和这个组件的可视代表（又称作一个 UI 代表）分离开。虽然可以实现定制的 UI 代表（参见 3.2.6 节中定制 UI 代表的例子），但在实际中很少实现定制的 UI 代表，因为组件的可视代表由当前的界面样式来决定。

另一方面，显示文本组件内容的方式与界面样式无关。再有，经常需要修改文本内容的可视代表但不改变这个组件的界面样式。例如，一个程序员的编辑器可能有一个选项，这个选项用彩色来显示程序语言的关键字。

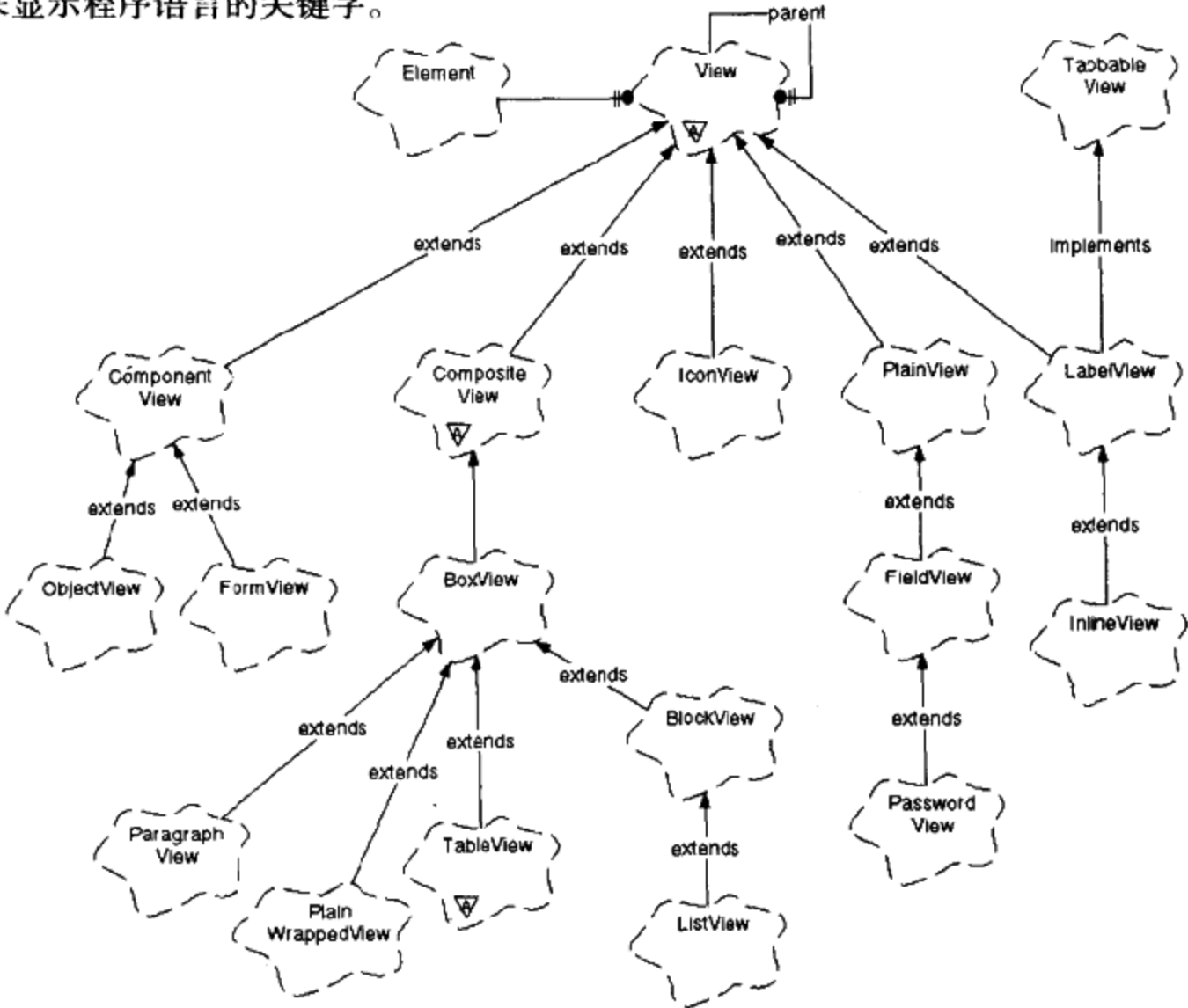


图 23-5 视图的分层类图

因为经常要修改文本组件的可视化代表，所以文本组件除有一个 UI 代表外，还有一个视图。文本视图允许在不替换这个组件的 UI 代表的情况下定制文本组件内容的可视化代表。

Swing 的文本包提供了大量的视图，如图 23-5 所示。

视图由抽象 View 类来定义，这个抽象 View 类维护对一个元素的引用。要了解被标准 Swing 文本组件使用的视图的列表，请参见表 23-1。

实现定制视图

如在 23.1 节“概述”中所介绍的，单行文本域和多行文本域的视图由组件的 UI 代表创建。因此，不实现一个定制的 UI 代表就不能为单行文本域和多行文本域指定一个定制视图。

另一方面，编辑窗格和文本窗格视图由组件的编辑器工具包来创建，因此编辑窗格和文本窗格的视图可以被定制（具有一个定制编辑器工具包的编辑器）而不需修改组件的 UI 代表。

图 23-6 所示的应用程序包含一个编辑窗格，这个窗格利用定制编辑器工具包来配备一个定制视图。这个视图提供了位置（包含在编辑器窗格的文档中）的图形代表。这个应用程序还提供了一个按钮，以便在加字符位置上插入一个位置。

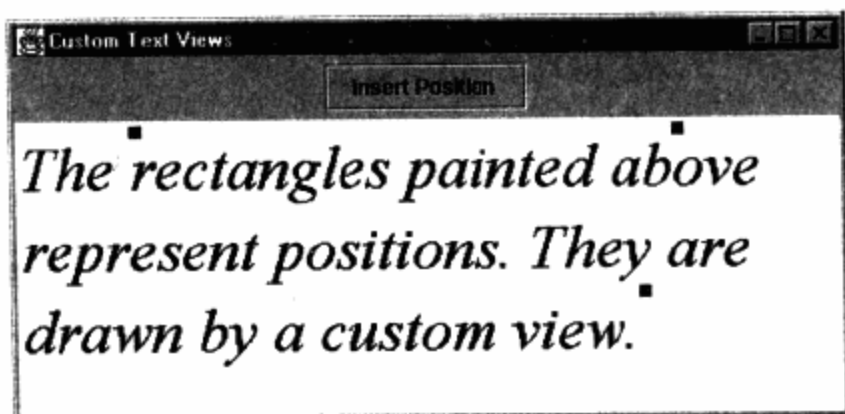


图 23-6 一个绘制位置的定制视图

这个应用程序创建一个 JEditorPane 实例，并把这个编辑器窗格的编辑器工具包设置为 CustomEditorKit 的一个实例。因为编辑器窗格和文本窗格的视图是由组件的编辑器工具包创建的，所以要为这个编辑器窗格指定一个定制编辑器工具包。

```
public class Test extends JFrame {
    JEditorPane editorPane = new JEditorPane ();
    Vector positions = new Vector ();
    Position.Bias bias = Position.Bias.Forward;
    ...
    public Test () {
        JPanel panel = new JPanel ();
        JButton button = new JButton ("Insert Position");
        Container contentPane = getContentPane ();
        panel.add (button);

        editorPane.setEditorKit (new CustomEditorKit ());
        editorPane.setFont (new Font ("Serif", Font.ITALIC, 36));
        ...
    }
}
```

CustomEditorKit 类是 DefaultEditorKit 的一个扩展，这个类还实现 ViewFactory 接口。这个编辑器工具包的 getViewFactory 方法返回对这个编辑器工具包自身的一个引用，而且这个编辑器工具包的 create 方法返回 CustomView 的一个实例。

```
class CustomEditorKit extends DefaultEditorKit
    implements ViewFactory {
    public ViewFactory getViewFactory () {
        return this;
    }
}
```

```

    |
    public View create (Element elem) {
        return new CustomView (elem);
    }
};

```

CustomView 类扩展 WrappedPlainView 并重载 paint 方法，paint 方法调用 super.paint () 来绘制这个编辑器窗格的内容。CustomView.paint 随后绘制每一个位置，通过激活这个小应用程序的按钮来创建这些位置。

```

class CustomView extends WrappedPlainView {
    public CustomView (Element elem) {
        super (elem);
    }
    public void paint (Graphics g, Shape a) {
        super.paint (g, a);

        Enumeration e = positions.elements ();
        Position p;
        while (e.hasMoreElements ()) {
            try {
                p = (Position) e.nextElement ();
                int offset = p.getOffset ();

                Shape shape = modelToView (Offset a, bias);
                Rectangle r = shape.getBounds ();

                g.setColor (Color.black);
                g.drawRect (r.x, r.y, 6, 6);

                g.setColor (Color.red);
                g.fillRect (r.x + 1, r.y + 1, 5, 5);
            }
            catch (BadLocationException ex) {
                ex.printStackTrace ();
            }
        }
    }
};

```

例 23-3 列出了图 23-6 所示应用程序的完整代码。

例 23-3 实现一个定制视图

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Test extends JFrame {
    JEditorPane editorPane = new JEditorPane ();
    Vector positions = new Vector ();
    Position.Bias bias = Position.Bias.Forward;

    class CustomView extends WrappedPlainView {
        public CustomView (Element elem) {
            super (elem);

```

```

    |
    public void paint (Graphics g, Shape a) {
        super.paint (g, a);

        Enumeration e = positions.elements ();
        Position p;

        while (e.hasMoreElements ()) {
            try {
                p = (Position) e.nextElement ();
                int offset = p.getOffset ();

                Shape shape = modelToView (offset a, bias);
                Rectangle r = shape.getBounds ();

                g.setColor (Color.black);
                g.drawRect (r.x, r.y, 6, 6);

                g.setColor (Color.red);
                g.fillRect (r.x + 1, r.y + 1, 5, 5);
            }
            catch (BadLocationException ex) {
                ex.printStackTrace ();
            }
        }
    }
};

class CustomEditorKit extends DefaultEditorKit
    implements ViewFactory {
    public ViewFactory getViewFactory () {
        return this;
    }

    public View create (Element elem) {
        return new CustomView (elem);
    }
}

public Test () {
    JPanel panel = new JPanel ();
    JButton button = new JButton ("Insert Position");
    Container contentPane = getContentPane ();

    panel.add (button);

    editorPane.setEditorKit (new CustomEditorKit ());
    editorPane.setFont (new Font ("Serif", Font.ITALIC, 36));

    contentPane.add (panel, BorderLayout.NORTH);
    contentPane.add (editorPane, BorderLayout.CENTER);

    button.addActionListener (new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            try {
                int p = editorPane.getCaretPosition ();
                Document document =
                    editorPane.getDocument ();

                positions.addElement (document.createPosition (p));
                editorPane.repaint ();
            }
            catch (BadLocationException ex) {

```

```

        ex.printStackTrace();
    }
}

public static void main (String args []) {
    GJApp.launch (new Test (),
        "Custom Text Views", 300, 300, 450, 300);
}

```

Swing 提示

文本视图

Swing 的轻量组件有一个 UI 代表，这个 UI 代表提供一个组件的界面样式。另外，有些组件（如列表框、组合框、树和表格等）还有单元绘制器，用这些绘制器来定制在组件单元中显示的对象的外观。这些单元绘制器独立于组件的界面样式。例如，可以不考虑当前的界面样式就安装一个定制单元绘制器，而且这些单元绘制器也能在任何界面样式下工作。

Swing 的文本组件没有单元，因此也没有单元绘制器。然而，与有单元绘制器的组件一样，Swing 的文本组件确实有需要经常被定制内容（但不改变界面样式）。为了提供在不改变界面样式的情况下定制文本组件内容的能力，这些组件除有 UI 代表外还有视图。

23.5 风格和风格的相关内容

可以为一个文本组件的内容范围指定一个简单的属性集，如 23.2 节“属性集和风格常量”中所介绍的。对更复杂的定制文本组件而言，更喜欢用从风格相关内容中获得的风格来设置文档的属性。本节给出了一个使用风格来设置文本窗格的段落属性的应用程序。

可简单地称风格为属性集，当修改它们时，它们将激发变化事件。风格有些特殊，因为不能直接实例化风格，而必须用 `StyledContext.addStyle` 方法来创建风格。

图 23-7 所示的应用程序包含一个文本窗格和一个菜单，这个菜单把段落属性设置为 title 或 body 段落风格。这个应用程序还显示与段落（包含在这个应用程序状态区中的加字符）相关联的风格。

图 23-7 的左上角图片显示了在启动这个应用程序且加字符已放在文本第一行后这个应用程序的样子。中间的图片 and 左下角的图片显示把第一段的属性设置为 title 后这个应用程序的样子。图 23-7 中两个右边的图片显示把第二段的风格设置为 body 后这个应用程序的样子。右上角的图片显示了加字符在第二段落中这个应用程序的样子，右下角图片显示了加字符在第三段落中这个应用程序的样子。注意，更新这个应用程序的状态区以显示所选段落的风格，第二段落的风格是 title，第三段落的风格是 default。

这个应用程序创建一个 `JTextPane` 实例并把编辑器工具包设置为 `CharacterEditorKit` 的一个实例。把文本窗格的动作装载到一个哈希表里，以便可以通过名字来获取这些动作。有关动作及通过名字来访问动作的更多信息，请参见 21.2.2 “动作和编辑器工具包”。

这个应用程序读名为 `text.txt` 的文本文件中的内容，这个文本窗格包裹在一个滚动窗格中，并把这个文本窗格指定为这个应用程序内容窗格的中心组件。然后，用一个专用的 `createMenuBar` 方法来创建这个应用程序的菜单栏。

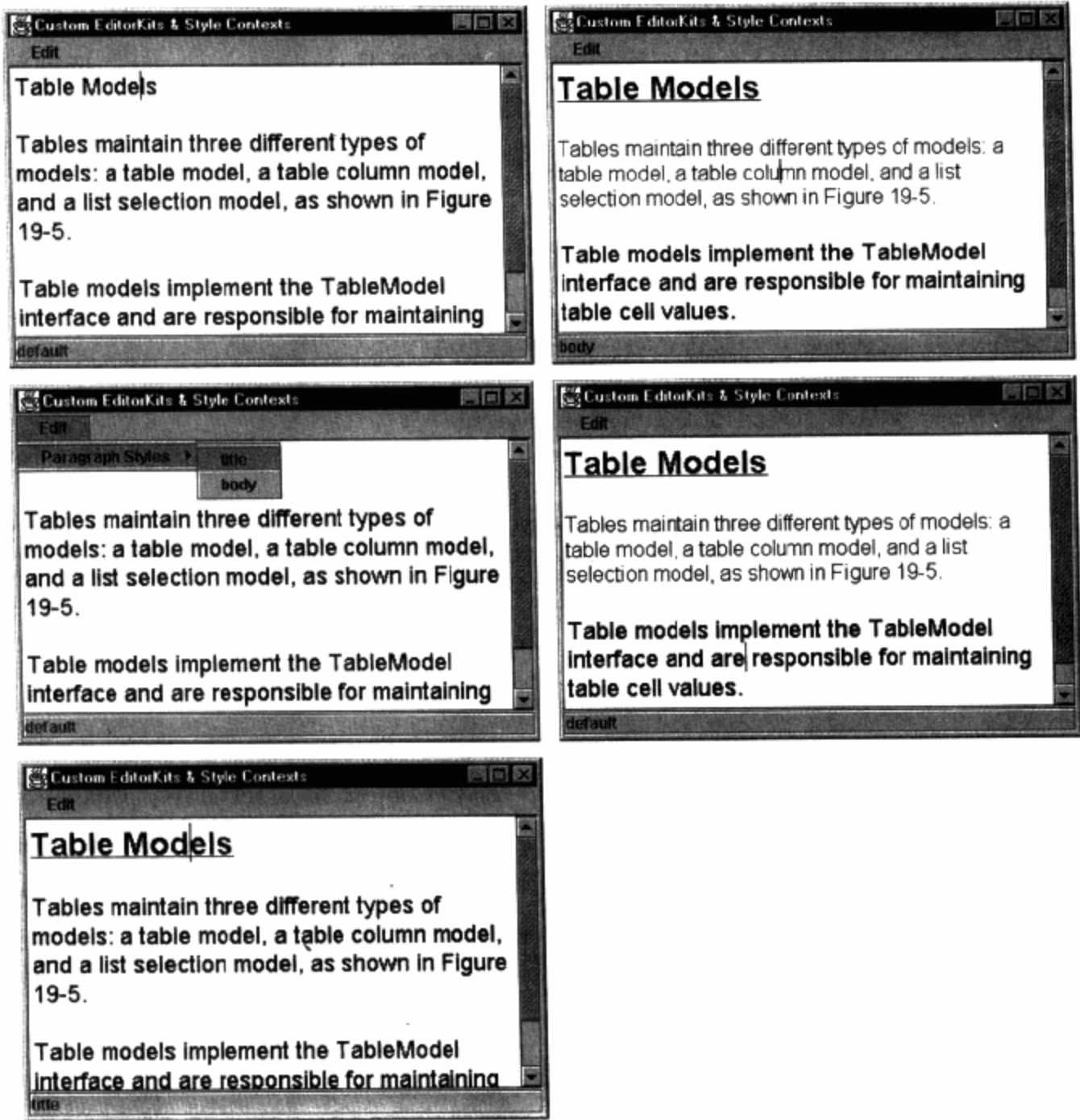


图 23-7 使用风格和风格的相关内容

```
public class Test extends JFrame {
    private JTextPane textPane = new JTextPane ();
    private Hashtable actionTable = new Hashtable ();
    private JCheckBoxMenuItem titleItem, bodyItem;

    public Test () {
        Container contentPane = getContentPane ();

        textPane.setEditorKit (new ChapterEditorKit ());
        textPane.setFont (new Font ("Dialog", Font.PLAIN, 18));

        // must load action table after setting editor kit ...
        loadActionTable ();

        readFile ("text.txt");

        contentPane.add (new JScrollPane (textPane),
                        BorderLayout.CENTER);
        contentPane.add (GJApp.getStatusArea (), BorderLayout.SOUTH);

        setJMenuBar (createMenuBar ());
    }
}
```


ChapterEditorKit 创建一个缺省的动作数组，添加这个数组到由编辑器工具包的超类提供的动作中 (StyledEditorKit)。当为这个应用程序的文本窗格调用 `getActions ()` 时，返回的 Action 数组是由 `getActions ()` 方法返回的数组。

```
class ChapterEditorKit extends StyledEditorKit {
    private CaretListener caretListener = new Listener ();
    private static ChapterStyleContext context =
        new ChapterStyleContext ();

    private static Action [] defaultActions = new Action [] {
        new ParagraphStyleAction (
            ChapterStyleContext.titleStyle,
            context.getStyle (ChapterStyleContext.titleStyle)),
        new ParagraphStyleAction (
            ChapterStyleContext.bodyStyle,
            context.getStyle (ChapterStyleContext.bodyStyle)),
    };

    public Action [] getActions () {
        return TextAction.augmentList (super.getActions (),
            defaultActions);
    }
    ...
}
```

从 ChapterEditorKit 的一个实例中获得由 ChapterStyleContext 指定的动作。ChapterStyleContext 是 StyleContext 类的一个扩展。ChapterStyleContext 定义 public 风格常量。ChapterStyleContext 的构造方法创建标题和正文风格，把这两种风格添加到缺省的风格（通过调用 `getStyle (DEFAULT_STYLE)` 来获得）中，这些缺省的风格由所有的编辑器工具包共享。

```
...
class ChapterStyleContext extends StyleContext {
    public static String titleStyle = "title",
        bodyStyle = "body";

    private static String [] defaultStyleNames = new String [] {
        new String (titleStyle),
        new String (bodyStyle) };

    public ChapterStyleContext () {
        Style root = getStyle (DEFAULT_STYLE);

        for (int i=0; i < defaultStyleNames.length; ++i) {
            String name = defaultStyleNames [i];
            Style s = addStyle (name, root);

            if (name.equals (titleStyle)) {
                StyleConstants.setFontFamily (s, "Dialog");
                StyleConstants.setFontSize (s, 24);
                StyleConstants.setBold (s, true);
                styleConstants.setUnderline (s, true);
            }
            else if (name.equals (bodyStyle)) {
                StyleConstants.setFontFamily (s, "Times-Roman");
                StyleConstants.setFontSize (s, 16);
            }
        }
    }
    ...
}
```

编辑器工具包有 `install` 和 `deinstall` 方法，当为一个指定的文本安装和卸载编辑器工具包时将分别调用这两个方法。`ChapterStyleContext` 类重载 `install` 方法以便把一个加字符监听器添加到这个工具包的文本组件中来更新这个应用程序的状态区。`deinstall` 方法删除这个监听器以便使文本组件处于安装编辑器工具包前的状态中。

这个加字符监听器获得加字符所在处的风格名字并据此来更新这个应用程序的状态区。

```
...
public void install (JEditorPane editorPane) {
    editorPane.addCaretListener (caretListener);
}
public void deinstall (JEditorPane editorPane) {
    editorPane.removeCaretListener (caretListener);
}
static class Listener implements CaretListener {
    public void caretUpdate (CaretEvent e) {
        int dot = e.getDot (), mark = e.getMark ();
        if (dot == mark) {
            JTextComponent c = (JTextComponent) e.getSource ();
            StyledDocument document =
                (StyledDocument) c.getDocument ();
            Element elem = document.getParagraphElement (dot);
            AttributeSet set = elem.getAttributes ();
            String name = (String) set.getAttribute (
                StyleConstants.NameAttribute);
            GJApp.showStatus (name);
        }
    }
}
...
```

用来为指定的段落设置属性的 `ParagraphStyleAction` 是 `StyledEditorKit.StyledTextAction` 的一个扩展。用一个名字和一个风格来构造这个动作。`actionPerformed` 方法为具有加字符的段落设置段落属性并用这个风格的名字来更新这个应用程序的状态区。

```
...
static class ParagraphStyleAction
    extends StyledEditorKit.StyledTextAction {
    private Style style;

    public ParagraphStyleAction (String nm, Style style) {
        super (nm);
        this.style = style;
    }

    public void actionPerformed (ActionEvent e) {
        setParagraphAttributes (getEditor (e), style, false);
        GJApp.showStatus (style.getName ());
    }
}
...
```

例 23-4 列出了图 23-7 所示应用程序的完整代码。

例 23-4 使用风格和风格的相关内容

```
import java.io.File;
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.FileReader;

public class Test extends JFrame {
    private JTextPane textPane = new JTextPane ();
    private Hashtable actionTable = new Hashtable ();
    private JCheckBoxMenuItem titleItem, bodyItem;

    public Test () {
        Container contentPane = getContentPane ();

        textPane.setEditorKit (new ChapterEditorKit ());
        textPane.setFont (new Font ("Dialog", Font.PLAIN, 18));
        // must load action table after setting editor kit ...
        loadActionTable ();

        readFile ("text.txt");

        contentPane.add (new JScrollPane (textPane),
                        BorderLayout.CENTER);
        contentPane.add (GJApp.getStatusArea (), BorderLayout.SOUTH);

        setJMenuBar (createMenuBar ());
    }

    private JMenuBar createMenuBar () {
        JMenuBar menuBar = new JMenuBar ();
        JMenu editMenu = new JMenu ("Edit"),
            styleMenu = new JMenu ("Paragraph Styles");

        styleMenu.add (getAction (ChapterStyleContext.titleStyle));
        styleMenu.add (getAction (ChapterStyleContext.bodyStyle));

        editMenu.add (styleMenu);
        menuBar.add (editMenu);
        return menuBar;
    }

    private void readFile (String filename) {
        EditorKit kit = textPane.getEditorKit ();
        Document doc = textPane.getDocument ();

        try {
            kit.read (new FileReader (filename), doc, 0);
        }
        catch (Exception ex) {
            ex.printStackTrace ();
        }
    }

    private void loadActionTable () {
        Action [] actions = textPane.getActions ();

        for (int i = 0; i < actions.length; ++i) {
```

```

        actionTable.put (actions [i] .getValue (Action.NAME),
                        actions [i]);
    }

    private Action getAction (String name) {
        return (Action) actionTable.get (name);
    }

    public static void main (String args []) {
        GJApp.launch (new Test (),
                      "Custom EditorKits & Style Contexts",
                      300, 300, 650, 275);
    }
}

class ChapterEditorKit extends StyledEditorKit {
    private CaretListener caretListener = new Listener ();
    private static ChapterStyleContext context =
        new ChapterStyleContext ();

    private static Action [] defaultActions = new Action [] {
        new ParagraphStyleAction (
            ChapterStyleContext.titleStyle,
            context.getStyle (ChapterStyleContext.titleStyle)),
        new ParagraphStyleAction (
            ChapterStyleContext.bodyStyle,
            context.getStyle (ChapterStyleContext.bodyStyle)),
    };

    public Action [] getActions () {
        return TextAction.augmentList (super.getActions (),
                                       defaultActions);
    }

    public void install (JEditorPane editorPane) {
        editorPane.addCaretListener (caretListener);
    }

    public void deinstall (JEditorPane editorPane) {
        editorPane.removeCaretListener (caretListener);
    }

    static class Listener implements CaretListener {
        public void caretUpdate (CaretEvent e) {
            int dot = e.getDot (), mark = e.getMark ();

            if (dot == mark) {
                JTextComponent c = (JTextComponent) e.getSource ();
                StyledDocument document =
                    (StyledDocument) c.getDocument ();
                Element elem = document.getParagraphElement (dot);
                AttributeSet set = elem.getAttributes ();
                String name = (String) set.getAttribute (
                    StyleConstants.NameAttribute);

                GJApp.showStatus (name);
            }
        }
    }

    static class ParagraphStyleAction
        extends StyledEditorKit.StyledTextAction {
        private Style style;
    }
}

```

```

public PaintedStyleAction (String title, Style style) {
    super (title);
    this.style = style;
}

@Override public void actionPerformed (ActionEvent e) {
    setParagraphAttributes (getEditor (e).getTextArea ());
    GUI.showStatus ("style: " + style.getName ());
}

class ChapterStyleContext extends StyleContext {
    public static String titleStyle = "title",
        bodyStyle = "body";

    private static String[] defaultStyleNames = new String[] {
        new String (titleStyle),
        new String (bodyStyle) };

    public ChapterStyleContext () {
        Style root = getStyle (DEFAULT_STYLE);
        for (int i = 0; i < defaultStyleNames.length; i++) {
            String name = defaultStyleNames [i];
            Style s = addStyle (name, root);
            if (name.equals (titleStyle)) {
                StyleConstants.setFontFamily (s, "Dialog");
                StyleConstants.setFontSize (s, 24);
                StyleConstants.setBold (s, true);
                StyleConstants.setItalic (s, true);
            } else if (name.equals (bodyStyle)) {
                StyleConstants.setFontFamily (s, "Times-Roman");
                StyleConstants.setFontSize (s, 16);
            }
        }
    }
}

```

23.6 元素

文档除维护一个文本组件的内容外，还维护一个元素的层次表。元素是对象，这些对象维护文档内的起始位置和结束位置，并维护起始位置和结束位置之间内容的属性集。图 23-8 示出了在 `javax.swing.text` 包中定义的元素类的一个类层次。

元素接口由 `AbstractDocument.AbstractElement` 类实现，这个类由 `AbstractDocument.BranchElement` 类和 `AbstractDocument.LeafElement` 类扩展。树根元素可以有子元素，但树叶元素没有子元素。`BranchElement` 类由 `DefaultStyledDocument.BranchElement` 和 `HTMLDocument.BlockElement` 来扩展。`LeafElement` 类由 `HTMLDocument.RunElement` 来扩展。

图 23-9 所示的应用程序包含一个分隔窗格，这个窗格有两个面板，左边的面板中有单行文本域、多行文本域、文本窗格和编辑器窗格，右边面板是 `ElementTreePanel` 的一个实例。`ElementTreePanel` 类使用一个树来显示文档的元素结构。`ElementTreePanel` 类被包含在 Swing 的示例中，因此在这里没有列出来。

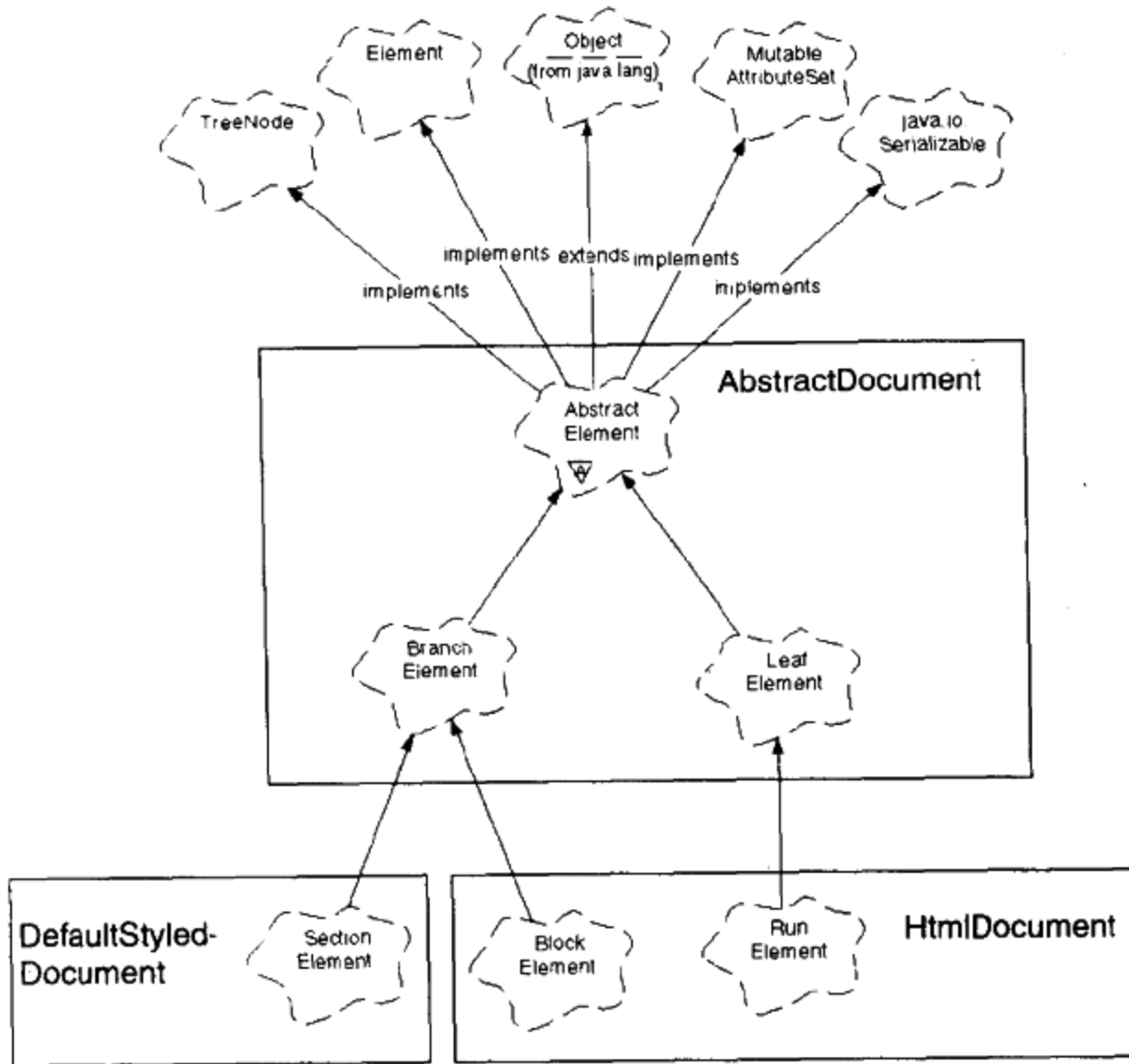


图 23-8 文档的类图

图 23-9 所示的四个图片说明了每个 Swing 文本组件的元素层次。单行文本域有一个最简单的元素层次，即有一个有内容的段落。图 23-9 最下面的图片显示一个编辑器窗格（显示一个 HTML 文档）的元素层次，而且到目前为止，编辑器窗格有最复杂元素层次。对文本组件的所有使用而言，缺省的元素层次已经完全够用了。

例 23-5 列出了图 23-9 所示应用程序的代码。

例 23-5 元素层次

```

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

public class Test extends JFrame {
    private JTextComponent components [] = new JTextComponent [] {
        new JTextField ("initial content"), new JTextArea (10, 20),
        new JTextPane (), new JEditorPane (),
    };
    private String borderNames [] = new String [] {
        "JTextField", "JTextArea", "JTextPane", "JEditorPane"
    };
    private JPanel textComponentPanel = new JPanel ();
}
  
```

```

private ElementTreePanel treePanel =
    new ElementTreePanel (components [0]);
private JSplitPane sp = new JSplitPane (
    JSplitPane.HORIZONTAL_SPLIT,
    new JScrollPane (textComponentPanel),
    new JScrollPane (treePanel));

public Test () {
    Container contentPane = getContentPane ();
    CaretListener listener = new Listener ();
    textComponentPanel.setBorder (
        BorderFactory.createTitledBorder ("Text Components"));
    for (int i=0; i < components.length; ++i) {
        JTextComponent c = (JTextComponent) components [i];
        c.addCaretListener (listener);
        c.setBorder (
            BorderFactory.createTitledBorder (borderNames [i]));
        if (i != 0) // JTextField
            readFile (c, "text.txt");
        if (i == 3) // JEditorPane
            JEditorPane
                editorPane = (JEditorPane) c;
        String url = "file:" +
            System.getProperty ("user.dir") +
            System.getProperty ("file.separator") +
            "java.util.Hashtable.html";
        editorPane.setEditable (false);
        try {
            editorPane.setPage (url);
        }
        catch (Exception ex) { ex.printStackTrace (); }
    }
    JScrollPane sp = new JScrollPane (c);
    sp.setPreferredSize (new Dimension (450, 450));
    panel.add (sp);
    |
    else
        panel.add (c);
    |
    sp.setDividerLocation (600);
    contentPane.add (sp, BorderLayout.CENTER);
    |
    class Listener implements CaretListener {
        public void caretUpdate (CaretEvent e) {
            JTextComponent c = (JTextComponent) e.getSource ();
            if (c != treePanel.getEditor ()) {
                sp.setRightComponent (treePanel =
                    new ElementTreePanel (c));
            }
        }
    }
    |
    private void readFile (JTextComponent c, String filename) {

```



```
try {
    c.read (new FileReader (filename), null);
}
catch (Exception ex) {
    ex.printStackTrace ();
}

public static void main (String args []) {
    GJApp.launch (new Test (),
        "Element Hierarchies", 300, 300, 800, 300);
}
```

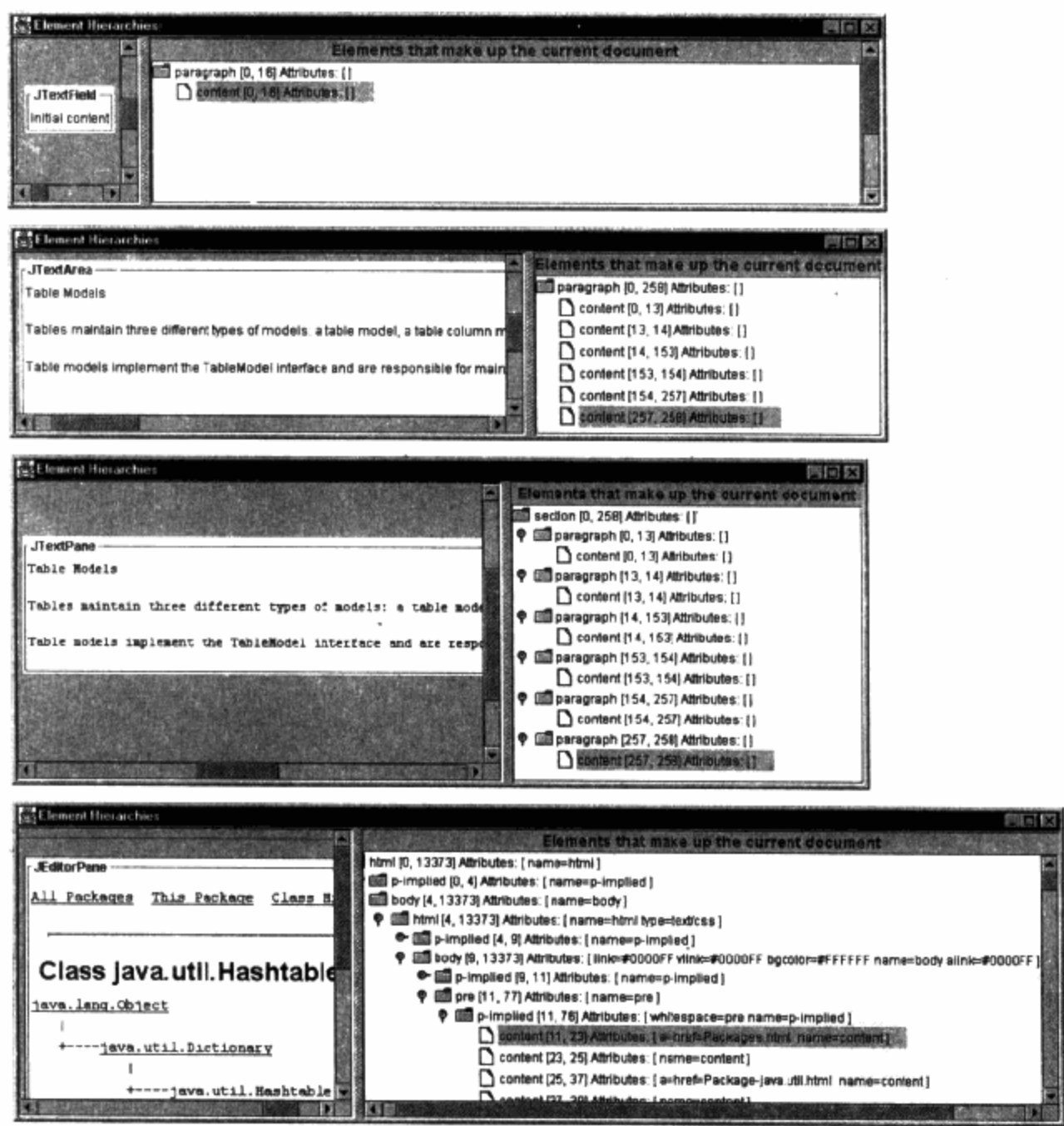


图 23-9 元素的层次

23.7 本章回顾

Swing 的文本包包含许多类和接口，其中大多数是内部类。类和接口的数量可能使学习 Swing 文本的开发人员感到害怕。为了在学习过程中提供帮助，本章给出了很多代码样例，这些样例使用了在 Swing 文本组件包中定义的最重要的类和接口。

第三部分 附录

附录 A 类图

- 类图图解
- 一个类图的例子

类图对软件开发者来说就像蓝图对于建筑一样重要。若要与别人简明扼要地交流自己的设计,类图是必不可少的,因此,本书大量使用类图来文档化 Java 基础类。

本书中的类图都是 Booch 类型的。对那些不熟悉 Booch 图的读者来说,下面我们要介绍一个摘要,并给出一个简单、但却相当完整的类图。

A.1 类图图解

下面是在类图中使用的元素的图解。

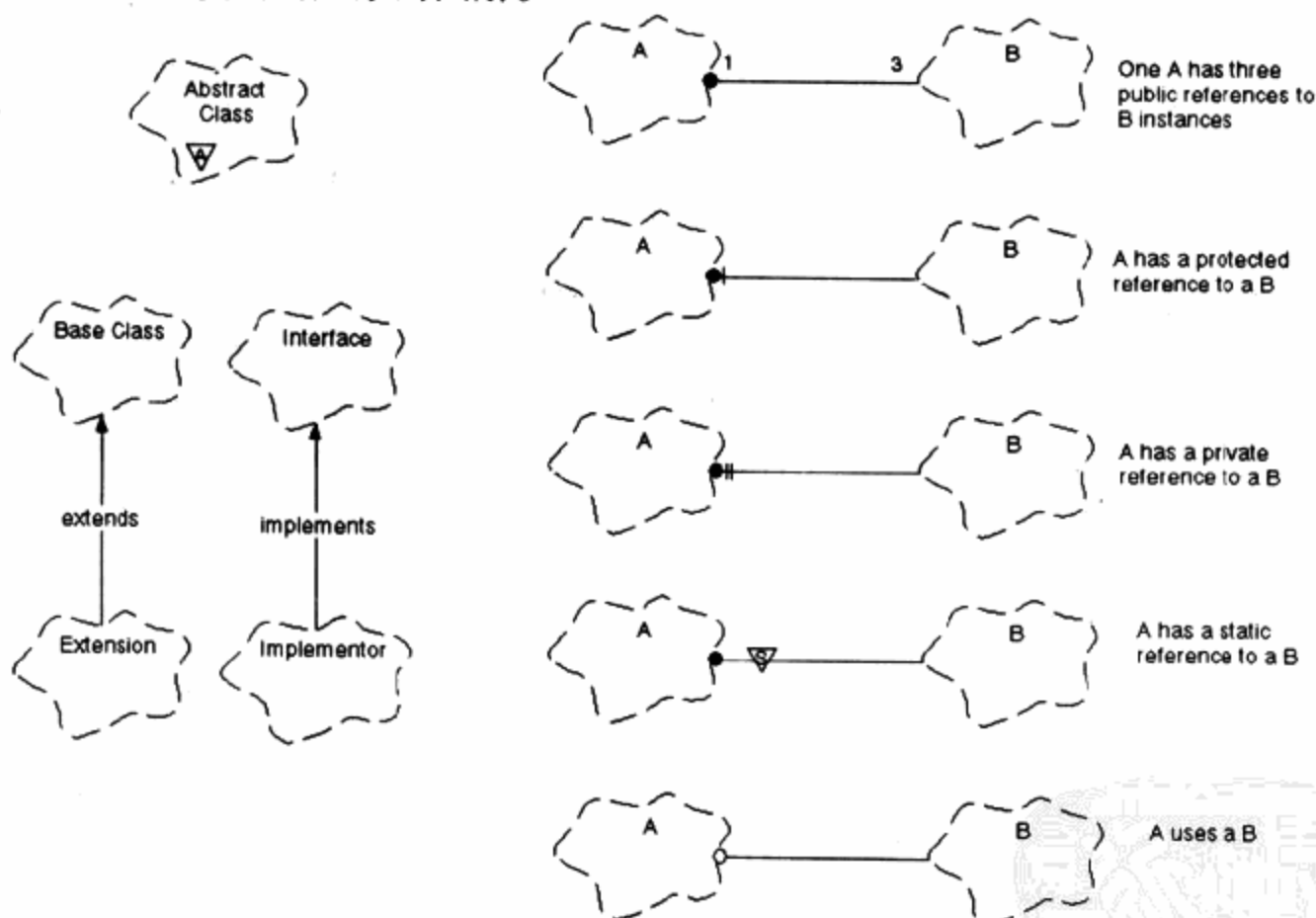


图 A-1 元素的图解

A.2 一个类图的例子

下面是 JScrollBar 类的类图。

JScrollBar 类扩展抽象的 JComponent 类并实现两个接口: Adjustable 和 Accessible。注意, Adjustable 和 Accessible 接口被定义为接口而不是类,这是因为 JScrollBar 实现这些接口。换句话说,除标识它们是接口外,对接口本身没有做任何工作。

JScrollBar 类维护对一个字符串的一个 static protected 引用。因为这个字符串的角色从这个

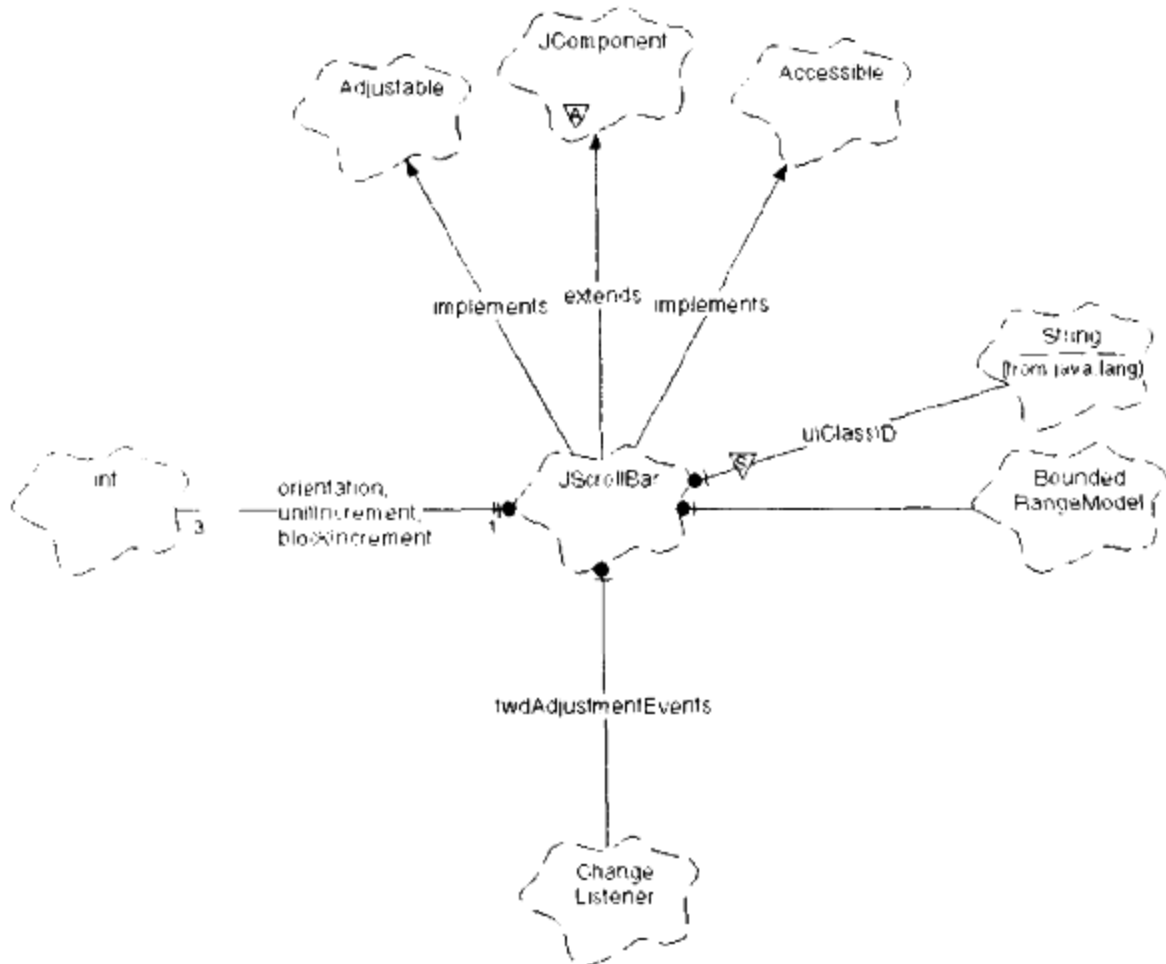


图 A-2 JScrollBar 类的类

类的名字中看不出来,所以用一个名字来强调 `JScrollBar` 和这个字符串之间的关系,这个名字阐明了这种关系(`uiClassID`)。

`JScrollBar` 还维护三个 `private integer` 值:滚动条的方向、单元增量和块增量。整数等内部类型在类图中是作为类来描述的。

`JScrollBar` 还维护对 `BoundedRangeModel` 和 `ChangeListener` 的实例的 `protected` 引用。`BoundedRangeModel` 是滚动条的模型,因此这种关系不包括一个标识符。`ChangeListener` 的角色从类名中是看不出来的,因此 `JScrollBar` 和 `ChangeListener` 之间的关系被确定为 `fwdAdjustmentEvents`。在代码中,关系标识符与引用名相同;在这里,滚动条的变化监听器把调整事件传递给它自己的监听器。

附录 B 插入式界面样式常量

表 B-1 列出了由 `javax.swing.plaf.basic.BasicLookAndFeel` 类定义的常量。这些常量被用来作为 Swing 组件界面样式的缺省值。有关表 B-1 列除的这些常量的使用情况,请参见“插入式界面样式”。

表 B-1 UIManager 常量^①

字符串常量	缺省值
<code>Button.font</code>	<code>dialogPlain12</code>
<code>Button.background</code>	<code>control</code>
<code>Button.foreground</code>	<code>controlText</code>
<code>Button.border</code>	<code>buttonBorder</code>
<code>Button.margin</code>	<code>new InsetsUIResource(2,14,2,14)</code>
<code>Button.textIconGap</code>	<code>new Integer(4)</code>
<code>Button.textShiftOffset</code>	<code>new Integer(0)</code>
<code>ToggleButton.font</code>	<code>dialogPlain12</code>
<code>ToggleButton.background</code>	<code>control</code>
<code>ToggleButton.foreground</code>	<code>controlText</code>
<code>ToggleButton.border</code>	<code>buttonToggleButtonBorder</code>
<code>ToggleButton.margin</code>	<code>new InsetsUIResource(2,14,2,14)</code>
<code>ToggleButton.textIconGap</code>	<code>new Integer(4)</code>
<code>ToggleButton.textShiftOffset</code>	<code>new Integer(0)</code>
<code>RadioButton.font</code>	<code>dialogPlain12</code>
<code>RadioButton.background</code>	<code>control</code>
<code>RadioButton.foreground</code>	<code>controlText</code>
<code>RadioButton.border</code>	<code>radioButtonBorder</code>
<code>RadioButton.margin</code>	<code>new InsetsUIResource(2,2,2,2)</code>
<code>RadioButton.textIconGap</code>	<code>new Integer(4)</code>
<code>RadioButton.textShiftOffset</code>	<code>new Integer(0)</code>
<code>RadioButton.icon</code>	<code>radioButtonIcon</code>
<code>CheckBox.font</code>	<code>dialogPlain12</code>
<code>CheckBox.background</code>	<code>control</code>
<code>CheckBox.foreground</code>	<code>controlText</code>
<code>CheckBox.border</code>	<code>radioButtonBorder</code>
<code>CheckBox.margin</code>	<code>new InsetsUIResource(2,2,2,2)</code>
<code>CheckBox.textIconGap</code>	<code>new Integer(4)</code>
<code>CheckBox.textShiftOffset</code>	<code>new Integer(0)</code>
<code>CheckBox.icon</code>	<code>CheckBoxIcon</code>
<code>ColorChooser.font</code>	<code>dialogPlain12</code>
<code>ColorChooser.background</code>	<code>control</code>
<code>ColorChooser.foreground</code>	<code>controlText</code>
<code>ColorChooser.swatchesSwatchSize</code>	<code>new Dimension(10,10)</code>

(续)

字符串常量	缺省值
ColorChooser.swatchesRecentSwatchSize	new Dimension(10,10)
ColorChooser.swatchesDefaultRecentColor	<i>control</i>
ColorChooser.rgbRedMnemonic	new Integer(KeyEvent.VK_R)
ColorChooser.rgbGreenMnemonic	new Integer(KeyEvent.VK_G)
ColorChooser.rgbBlueMnemonic	new Integer(KeyEvent.VK_B)
ComboBox.font	dialogPlain12
ComboBox.background	white
ComboBox.foreground	black
ComboBox.selectionBackground	<i>textHighlight</i>
ComboBox.selectionForeground	<i>textHighlightText</i>
ComboBox.disabledBackground	<i>control</i>
ComboBox.disabledForeground	<i>textInactiveText</i>
FileChooser.cancelButtonMnemonic	new Integer(KeyEvent.VK_C)
FileChooser.saveButtonMnemonic	new Integer(KeyEvent.VK_S)
FileChooser.openButtonMnemonic	new Integer(KeyEvent.VK_O)
FileChooser.updateButtonMnemonic	new Integer(KeyEvent.VK_U)
FileChooser.helpButtonMnemonic	new Integer(KeyEvent.VK_H)
FileChooser.newFolderIcon	newFolderIcon
FileChooser.upFolderIcon	upFolderIcon
FileChooser.homeFolderIcon	homeFolderIcon
FileChooser.detailsViewIcon	detailsViewIcon
FileChooser.listViewIcon	listViewIcon
FileView.directoryIcon	directoryIcon
FileView.fileIcon	fileIcon
FileView.computerIcon	computerIcon
FileView.hardDriveIcon	hardDriveIcon
FileView.floppyDriveIcon	floppyDriveIcon
InternalFrame.titleFont	dialogBold12
InternalFrame.border	internalFrameBorder
InternalFrame.icon	"icons/JavaCup.gif"
InternalFrame.maximizeIcon	BasicIconFactory.createEmptyFrameIcon()
InternalFrame.minimizeIcon	BasicIconFactory.createEmptyFrameIcon()
InternalFrame.iconifyIcon	BasicIconFactory.createEmptyFrameIcon()
InternalFrame.closeIcon	BasicIconFactory.createEmptyFrameIcon()
InternalFrame.activeTitleBackground	<i>activeCaption</i>
InternalFrame.activeTitleForeground	<i>activeCaptionText</i>
InternalFrame.inactiveTitleBackground	<i>inactiveCaption</i>
InternalFrame.inactiveTitleForeground	<i>inactiveCaptionText</i>
DesktopIcon.border	internalFrameBorder
Desktop.background	<i>desktop</i>
Label.font	dialogPlain12
Label.background	<i>control</i>
Label.foreground	<i>controlText</i>
Label.disabledforeground	white
Label.disabledShadow	<i>controlShadow</i>

(续)

字符串常量	缺省值
Label.border	null
List.font	dialogPlain12
List.background	window
List.foreground	textText
List.selectionBackground	textHighlight
List.selectionForeground	textHighlightText
List.focusCellHighlightBorder	focusCellHighlightBorder
List.border	null
List.cellRenderer	listCellRendererActiveValue
MenuBar.font	dialogPlain12
MenuBar.background	menu
MenuBar.foreground	menuText
MenuBar.border	menuBarBorder
MenuItem.font	dialogPlain12
MenuItem.acceleratorFont	dialogPlain12
MenuItem.background	menu
MenuItem.foreground	menuText
MenuItem.selectionForeground	textHighlightText
MenuItem.selectionBackground	textHighlight
MenuItem.disabledForeground	null
MenuItem.acceleratorForeground	menuText
MenuItem.acceleratorSelectionForeground	textHighlightText
MenuItem.border	marginBorder
MenuItem.borderPainted	Boolean.FALSE
MenuItem.margin	new InsetsUIResource(2,2,2,2)
MenuItem.checkIcon	menuItemCheckIcon
MenuItem.arrowIcon	menuItemArrowIcon
RadioButtonMenuItem.font	dialogPlain12
RadioButtonMenuItem.acceleratorFont	dialogPlain12
RadioButtonMenuItem.background	menu
RadioButtonMenuItem.foreground	menuText
RadioButtonMenuItem.selectionForeground	textHighlightText
RadioButtonMenuItem.selectionBackground	textHighlight
RadioButtonMenuItem.disabledForeground	null
RadioButtonMenuItem.acceleratorForeground	menuText
RadioButtonMenuItem.acceleratorSelectionForeground	textHighlightText
RadioButtonMenuItem.border	marginBorder
RadioButtonMenuItem.borderPainted	Boolean.FALSE
RadioButtonMenuItem.margin	new InsetsUIResource(2,2,2,2)
RadioButtonMenuItem.checkIcon	radioButtonMenuItemIcon
RadioButtonMenuItem.arrowIcon	menuItemArrowIcon
CheckBoxMenuItem.font	dialogPlain12
CheckBoxMenuItem.acceleratorFont	dialogPlain12
CheckBoxMenuItem.background	menu

(续)

字符串常量	缺省值
CheckBoxMenuItem.foreground	<i>menuText</i>
CheckBoxMenuItem.selectionForeground	<i>textHighlightText</i>
CheckBoxMenuItem.selectionBackground	<i>textHighlight</i>
CheckBoxMenuItem.disabledForeground	null
CheckBoxMenuItem.acceleratorForeground	<i>menuText</i>
CheckBoxMenuItem. acceleratorSelectionForeground	<i>textHighlightText</i>
CheckBoxMenuItem.border	marginBorder
CheckBoxMenuItem.borderPainted	Boolean.FALSE
CheckBoxMenuItem.margin	new InsetsUIResource(2,2,2,2)
CheckBoxMenuItem.checkIcon	checkBoxMenuItemIcon
CheckBoxMenuItem.arrowIcon	menuItemArrowIcon
Menu.font	dialogPlain12
Menu.acceleratorFont	dialogPlain12
Menu.background	<i>menu</i>
Menu.foreground	<i>menuText</i>
Menu.selectionForeground	<i>textHighlightText</i>
Menu.selectionBackground	<i>textHighlight</i>
Menu.disabledForeground	null
Menu.acceleratorForeground	<i>menuText</i>
Menu.acceleratorSelectionForeground	<i>textHighlightText</i>
Menu.border	marginBorder
Menu.borderPainted	Boolean.FALSE
Menu.margin	new InsetsUIResource(2,2,2,2)
Menu.checkIcon	menuItemCheckIcon
Menu.arrowIcon	menuItemArrowIcon
Menu.consumesTabs	Boolean.TRUE
PopupMenu.font	dialogPlain12
PopupMenu.background	<i>menu</i>
PopupMenu.foreground	<i>menuText</i>
PopupMenu.border	raisedBevelBorder
OptionPane.font	dialogPlain12
OptionPane.background	<i>control</i>
OptionPane.foreground	<i>controlText</i>
OptionPane.messageForeground	<i>controlText</i>
OptionPane.border	optionPaneBorder
OptionPane.messageAreaBorder	zeroBorder
OptionPane.buttonAreaBorder	optionPaneButtonAreaBorder
OptionPane.minimumSize	optionPaneMinimumSize
OptionPane.errorIcon	"icons/Error.gif"
OptionPane.informationIcon	"icons/Inform.gif"
OptionPane.warningIcon	"icons/Warn.gif"
OptionPane.questionIcon	"icons/Question.gif"
Panel.font	dialogPlain12
Panel.background	<i>control</i>

(续)

字符串常量	缺省值
Panel, foreground	<i>textText</i>
ProgressBar, font	<i>dialogPlain12</i>
ProgressBar, foreground	<i>textHighlight</i>
ProgressBar, background	<i>control</i>
ProgressBar, selectionForeground	<i>control</i>
ProgressBar, selectionBackground	<i>textHighlight</i>
ProgressBar, border	<i>progressBarBorder</i>
ProgressBar, cellLength	<i>new Integer(1)</i>
ProgressBar, cellSpacing	<i>new Integer(0)</i>
Separator, shadow	<i>controlShadow</i>
Separator, highlight	<i>controlLtHighlight</i>
ScrollBar, background	<i>scrollBarTrack</i>
ScrollBar, foreground	<i>control</i>
ScrollBar, track	<i>scrollbar</i>
ScrollBar, trackHighlight	<i>controlDkShadow</i>
ScrollBar, thumb	<i>control</i>
ScrollBar, thumbHighlight	<i>controlLtHighlight</i>
ScrollBar, thumbDarkShadow	<i>controlDkShadow</i>
ScrollBar, thumbLightShadow	<i>controlShadow</i>
ScrollBar, border	<i>null</i>
ScrollBar, minimumThumbSize	<i>minimumThumbSize</i>
ScrollBar, maximumThumbSize	<i>maximumThumbSize</i>
ScrollPane, font	<i>dialogPlain12</i>
ScrollPane, background	<i>control</i>
ScrollPane, foreground	<i>controlText</i>
ScrollPane, border	<i>etchedBorder</i>
ScrollPane, viewportBorder	<i>null</i>
Viewport, font	<i>dialogPlain12</i>
Viewport, background	<i>control</i>
Viewport, foreground	<i>textText</i>
Slider, foreground	<i>control</i>
Slider, background	<i>control</i>
Slider, highlight	<i>controlLtHighlight</i>
Slider, shadow	<i>controlShadow</i>
Slider, focus	<i>controlDkShadow</i>
Slider, border	<i>null</i>
Slider, focusInsets	<i>sliderFocusInsets</i>
SplitPane, background	<i>control</i>
SplitPane, highlight	<i>controlLtHighlight</i>
SplitPane, shadow	<i>controlShadow</i>
SplitPane, border	<i>splitPaneBorder</i>
SplitPane, dividerSize	<i>new Integer(5)</i>
TabbedPane, font	<i>dialogPlain12</i>
TabbedPane, background	<i>control</i>
TabbedPane, foreground	<i>controlText</i>

(续)

字符串常量	缺省值
TabbedPane.lightHighlight	<i>controlLtHighlight</i>
TabbedPane.highlight	<i>controlHighlight</i>
TabbedPane.shadow	<i>controlShadow</i>
TabbedPane.darkShadow	<i>controlDkShadow</i>
TabbedPane.focus	<i>controlText</i>
TabbedPane.textIconGap	<i>new Integer(4)</i>
TabbedPane.tabInsets	<i>tabbedPaneTabInsets</i>
TabbedPane.selectesTabPadInsets	<i>tabbedPaneTabPadInsets</i>
TabbedPane.tabAreaInsets	<i>tabbedPaneTabAreaInsets</i>
TabbedPane.contentBorderInsets	<i>tabbedPaneContentBorderInsets</i>
TabbedPane.tabRunOverlay	<i>new Integer(2)</i>
Table.font	<i>dialogPlain12</i>
Table.foreground	<i>controlText</i>
Table.background	<i>window</i>
Table.selectionForeground	<i>textHighlightText</i>
Table.selectionBackground	<i>textHighlight</i>
Table.gridColor	<i>gray</i>
Table.focusCellBackground	<i>window</i>
Table.focusCellForeground	<i>controlText</i>
Table.focusCellHighlightBorder	<i>focusCellHighlightBorder</i>
Table.scrollPaneBorder	<i>loweredBevelBorder</i>
TableHeader.font	<i>dialogplain 12</i>
TableHeader.foreground	<i>controlText</i>
TableHeader.background	<i>control</i>
TableHeader.cellBorder	<i>raisedBevelBorder</i>
TextField.font	<i>sansSerifPlain12</i>
TextField.background	<i>window</i>
TextField.foreground	<i>textText</i>
TextField.inactiveForeground	<i>textInactiveText</i>
TextField.selectionBackground	<i>textHighlight</i>
TextField.selectionForeground	<i>textHighlightText</i>
TextField.caretForeground	<i>textText</i>
TextField.caretBlinkRate	<i>createBlinkRate</i>
TextField.border	<i>textFieldBorder</i>
TextField.margin	<i>zeroInsets</i>
TextField.keyBindings	<i>fieldBindings</i>
PasswordField.font	<i>monospacedPlain12</i>
PasswordField.background	<i>window</i>
PasswordField.foreground	<i>textText</i>
PasswordField.inactiveForeground	<i>textInactiveText</i>
PasswordField.selectionBackground	<i>textHighlight</i>
PasswordField.selectionForeground	<i>textHighlightText</i>
PasswordField.caretForeground	<i>textText</i>
PasswordField.caretBlinkRate	<i>createBlinkRate</i>
PasswordField.border	<i>textFieldBorder</i>

(续)

字符串常量	缺省值
PasswordField.margin	zeroInsets
PasswordField.keyBindings	fieldBindings
TextArea.font	monospacedPlain12
TextArea.background	window
TextArea.foreground	textText
TextArea.inactiveForeground	textInactiveText
TextArea.selectionBackground	textHighlight
TextArea.selectionForeground	textHighlightText
TextArea.caretForeground	textText
TextArea.caretBlinkRate	creatBlinkRate
TextArea.border	marginBorder
TextArea.margin	zeroInsets
TextArea.keyBindings	multilineBindings
TextPane.font	serifPlain12
TextPane.background	white
TextPane.foreground	textText
TextPane.selectionBackground	lightGray
TextPane.selectionForeground	textHighlightText
TextPane.caretForeground	textText
TextPane.caretBlinkRate	createBlinkRate
TextPane.inactiveForeground	textInactiveText
TextPane.border	marginBorder
TextPane.margin	editorMargin
TextPane.keyBindings	multilineBindings
EditorPane.font	serifPlain12
EditorPane.background	white
EditorPane.foreground	textText
EditorPane.selectionBackground	lightGray
EditorPane.selectionForeground	textHighlightText
EditorPane.caretForeground	red
EditorPane.caretBlinkRate	createBlinkRate
EditorPane.inactiveForeground	textInactiveText
EditorPane.border	marginBorder
EditorPane.margin	editorMargin
EditorPane.keyBindings	multilineBindings
TitledBorder.font	dialogPlain12
TitledBorder.titleColor	controlText
TitledBorder.border	etchedBorder
ToolBar.font	dialogPlain12
ToolBar.background	control
ToolBar.foreground	controlText
ToolBar.dockingBackground	control
ToolBar.dockingForeground	red
ToolBar.floatingBackground	control
ToolBar.floatingForeground	darkGray

(续)

字符串常量	缺省值
ToolBar.border	etchedBorder
ToolBar.separatorSize	toolBarSeparatorSize
ToolTip.font	sansSerifPlain12
ToolTip.background	<i>info</i>
ToolTip.foreground	<i>infoText</i>
ToolTip.border	blackLineBorder
Tree.font	dialogPlain12
Tree.background	<i>window</i>
Tree.foreground	<i>textText</i>
Tree.hash	gray
Tree.textForeground	<i>textText</i>
Tree.textBackground	<i>text</i>
Tree.selectionForeground	<i>textHighlightText</i>
Tree.selectionBackground	<i>textHighlight</i>
Tree.selectionBorderColor	black
Tree.editorBorder	blackLineBorder
Tree.leftChildIndent	new Integer(7)
Tree.rightChildIndent	new Integer(13)
Tree.rowHeight	new Integer(16)
Tree.scrollsOnExpand	Boolean.TRUE
Tree.openIcon	"icons/TreeOpen.gif"
Tree.closedIcon	"icons/TreeClosed.gif"
Tree.leafIcon	"icons/TreeLeaf.gif"
Tree.expandedIcon	null
Tree.collapsedIcon	null
Tree.changeSelectionWithFocus	Boolean.TRUE
Tree.drawsFocusBorderAroundIcon	Boolean.FALSE

① 系统颜色常量以斜体字表示。